36th European Conference on Object-Oriented Programming

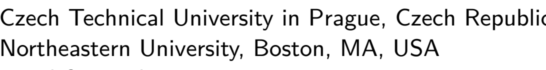
ECOOP 2022, June 6–10, 2022, Berlin, Germany

Edited by

Karim Ali
Jan Vitek

Editors

Karim Ali
University of Alberta, Canada
karim.ali@ualberta.ca

Jan Vitek

Czech Technical University in Prague, Czech Republic
Northeastern University, Boston, MA, USA
j.vitek@neu.edu

ACM Classification 2012
Software and its engineering

ISBN 978-3-95977-225-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-225-9.

Publication date
June, 2022

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2022.0

ISBN 978-3-95977-225-9 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

This volume is dedicated to the memory of Camil Demetrescu and Eelco Visser.

Contents

Message from the Program Chairs
Karim Ali and Jan Vitek 0:xi

Message from the Artifact Evaluation Chairs
Alessandra Gorla and Stefan Winter 0:xiii

Foreword by the President of AITO
Eric Jul 0:xv

Authors
.... 0:xvii–0:xx

Regular Papers

Verified Compilation and Optimization of Floating-Point Programs in CakeML
Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox 1:1–1:28

Elementary Type Inference
Jinxu Zhao and Bruno C. d. S. Oliveira 2:1–2:28

Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs
Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi 3:1–3:28

Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types
Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida 4:1–4:29

How to Take the Inverse of a Type
Daniel Marshall and Dominic Orchard 5:1–5:27

Compiling Volatile Correctly in Java
Shuyang Liu, John Bender, and Jens Palsberg 6:1–6:26

Functional Programming with Datalog
André Pacak and Sebastian Erdweg 7:1–7:28

Design-By-Contract for *Flexible* Multiparty Session Protocols
Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida ... 8:1–8:28

A Deterministic Memory Allocator for Dynamic Symbolic Execution
Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar 9:1–9:26

Accumulation Analysis
Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst 10:1–10:30

Concolic Execution for WebAssembly
Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão 11:1–11:29

Defining Corecursive Functions in Coq Using Approximations
Vlad Rusu and David Nowak 12:1–12:24

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

REST: Integrating Term Rewriting with Program Verification
Zachary Grannan, Niki Vazou, Eva Darulova, and Alexander J. Summers 13:1–13:29

Static Analysis for AWS Best Practices in Python Code
Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson 14:1–14:28

What If We Don't Pop the Stack? The Return of 2nd-Class Values
Anxhelo Xhebraj, Oliver Bračevac, Guannan Wei, and Tiark Rompf 15:1–15:29

Maniposynth: Bimodal Tangible Functional Programming
Brian Hempel and Ravi Chugh 16:1–16:29

Synchron – An API and Runtime for Embedded Systems
Abhiroop Sarkar, Bo Joel Svensson, and Mary Sheeran 17:1–17:29

Direct Foundations for Compositional Programming
Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira 18:1–18:28

Low-Level Bi-Abduction
Lukáš Holík, Petr Peringer, Adam Rogalewicz, Veronika Šoková, Tomáš Vojnar, and Florian Zuleger 19:1–19:30

Functional Programming for Distributed Systems with XC
Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli 20:1–20:28

PEDroid: Automatically Extracting Patches from Android App Updates
Hehao Li, Yizhuo Wang, Yiwei Zhang, Juanru Li, and Dawu Gu 21:1–21:31

Ferrite: A Judgmental Embedding of Session Types in Rust
Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho 22:1–22:28

A Self-Dual Distillation of Session Types (Pearl)
Jules Jacobs 23:1–23:22

JavaScript Sealed Classes
Manuel Serrano 24:1–24:27

Union Types with Disjoint Switches
Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira 25:1–25:31

Fair Termination of Multiparty Sessions
Luca Ciccone, Francesco Dagnino, and Luca Padovani 26:1–26:26

API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3 (Pearl)
Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença 27:1–27:28

Global Type Inference for Featherweight Generic Java
Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann 28:1–28:27

Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BackREST
François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams 29:1–29:30

QILIN: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis
Dongjie He, Jingbo Lu, and Jingling Xue 30:1–30:29

NWGraph: A Library of Generic Graph Algorithms and Data Structures in C++20
Andrew Lumsdaine, Luke D'Alessandro, Kevin Deweese, Jesun Firoz, Xu Tony Liu, Scott McMillan, John Phillip Ratzloff, and Marcin Zalewski 31:1–31:28

Extended Abstracts

Vincent: Green Hot Methods in the JVM
Kenan Liu , Khaled Mahmoud, Joonhwan Yoo, and Yu David Liu 32:1–32:30

Hinted Dictionaries: Efficient Functional Ordered Sets and Maps
Amir Shaikhha, Mahdi Ghorbani, and Hesam Shahrokhi 33:1–33:3

Slicing of Probabilistic Programs Based on Specifications
Marcelo Navarro and Federico Olmedo 34:1–34:2

Prisma: A Tierless Language for Enforcing Contract-Client Protocols in Decentralized Applications
David Richter, David Kretzler, Pascal Weisenburger, Guido Salvaneschi, Sebastian Faust, and Mira Mezini 35:1–35:4

Message from the Program Chairs

Started in 1987, ECOOP is Europe's oldest programming conference, welcoming papers on all practical and theoretical investigations of programming languages, systems and environment providing innovative solutions to real problems as well as evaluations of existing solutions. Papers were submitted to one of four categories: *Research* for papers that advance the state of the art in programming; *Reproduction* for empirical evaluations that reconstructs a published experiment in a different context in order to validate the results of that earlier work; *Experience* for applications of known techniques in practice; and *Pearl* for papers that either explain a known idea in an elegant way or unconventional papers introducing ideas that may take some time to substantiate. ECOOP is a selective venue, with acceptance, by traditio, capped at 25% of all submissions and re-submissions. The chairs thank the Program Committee: A. Donaldson, B. Hermann, M. Sridharan, S. Alimadadi, A. Bieniusa, S. Blackburn, S. Blazy, E. Brady, L.r Bulej, S. Chiba, A. Cohen, E. Darulova, W. De Meuter, D. Dreyer, S. Drossopoulou, S. Ducasse, S. Erdweg, S. Fowler, J. Franco, D. Garg, S. Gay, J. Gibbons, E. Gonzalez Boix, P. Haller, R. Hirschfeld, T. Hosking, D. Lea, M. Luján, M. Madsen, A. Møller, J. Noble, M. Odersky, B. C.d.S. Oliveira, K. Ostermann, T. Petricek, A. Potanin, M. Rapoport, M. Rigger, G. Salvaneschi, T. Schrijvers, M. Serrano, A. Silva, E. Tosch, L. Tratt, V. Vasconcelos, E. Visser, T. Wrigstad, T. Xie, J. Xue, E. Zucca. We thank the Extended Review Committee: Q. Stiévenart, C. Koparkar, K. Narasimhan, S. Singh, J. Yang, L. De Simone, M. Jimenez, T. Nakamaru, J. Immanuel Brachthäuser, O. Bračevac, J. Norlinder, D. He, C. Zhang, M. Kruliš, V. Dort, V. Horky, W. Ye, B. Rehman, K. Marussy, P. Koronkevich, H. Dang, A. Tondwalkar, I. Kabir, A. Renda, M. Chiari, O. Flückiger, P. Maj, C. Hsieh, M. Raab, M. Schröder, D. Justo, L. Schütze, P. Weisenburger, E. D'Osualdo, S. Keuchel, J. An, S. Keidel, P. Rein, T. Mattis, A. Gorla, S. Winter. This year saw a number of innovations:

- **Multiple rounds.** ECOOP has two main rounds of submissions per year. Each round supports both minor and major revisions. Major revisions are handled in the next round (either the same year or the next) by the same reviwers.
- **No format or length restrictions.** In order reduce friction for authors, papers can come in any format and at any length. This applies to submisisions, final versions must abide by the publisher's requirements.
- **Arfitacts and Papers together.** Every submitted paper can be accompanied with an artifact, submitted 10 days after the paper. Both submission are evaluated in parallel by overlapping committees as members of the artifact evaluation committee were invited to served on the conference review committee.
- **Journal First/Last.** Papers can be submittted either one of three associated journals and be invited to present at the meeting. Furthermore, some accepted papers can be forward to journals.

Overall, we found these innovations to have worked well. Clearly more experience is needed to draw any broader conclusions. We do encourage future chair to keep experimenting.

Karim Ali **Jan Vitek**

Message from the Artifact Evaluation Chairs

ECOOP has a long-standing tradition of offering artifact evaluation dating back to 2013. For the first time this year, though, the artifact evaluation process involved every single paper submission to ECOOP 2022, rather than just accepted papers, and happened in parallel with the paper review process. Besides providing feedback on the artifacts irrespective of paper acceptance, evaluation results were made available to the technical PC. Artifact submissions could thus provide more insights on the technical contributions described in the papers, and help to improve the overall review process.

To handle the higher review load that such a process entails, we recruited an artifact evaluation committee that was almost twice as large as for last year's ECOOP and included both experienced and novice artifact reviewers. The submission deadlines for artifacts were just 10 days after the paper deadlines for both submission rounds. We received a total of 57 submissions (39 for R1 and 18 for R2). After a kick-the tires review and author response phase, during which authors had the opportunity to clarify or address technical issues with their submissions, each submitted artifact was reviewed by at least three committee members, leading to an overall review load of 4–5 artifact reviews per committee member.

Following the positive experience with adopting ACM's artifact badges for ECOOP 2021, we adopted the same badging policies for ECOOP 2022. The artifact evaluation committee positively evaluated 46 submissions (33/13 for R1/R2) as *functional* or *reusable*, out of which 25 belong to papers to appear in the technical program of ECOOP 2022. Seven submitted artifacts (4/3 in R1/R2) that did not pass the bar for the *functional* and *reusable* badges, were found eligible for the *available* badge, 2 of which are associated with papers accepted for presentation at ECOOP 2022 (both from R1).

To streamline the artifact review process and to decouple artifact from paper review aspects, we asked authors to submit documentation of explicit *claims* in a pre-specified format that the artifact evaluation committee checked the artifacts against. At the same time, the PC could assess the *importance of these claims* for the submitted papers as a frame of reference for the strength of support for the paper that an artifact can provide. This separation greatly facilitated the artifact evaluation committee's discussions regarding which badges to award. The details of this process are documented in the call for artifacts (`https://doi.org/10.5281/zenodo.6553744`), the artifact submission template (`https://doi.org/10.5281/zenodo.5720714`), and an artifact review template (`https://doi.org/10.5281/zenodo.5750738`) that we provided as guidance for artifact reviewers in addition to prior community guidance linked from the call for artifacts.

The smooth and thorough artifact evaluation process would have not been possible without the 39 members of the committee, who handled the artifact review workload and contributed to the technical PC discussions with great dedication. For this reason, we would like to thank them for their valuable work and the inspiring discussions.

Alessandra Gorla
Artifact Evaluation Co-chair
IMDEA Software Institute

Stefan Winter
Artifact Evaluation Co-chair
Ludwig-Maximilians-Universität München

Foreword by the President of AITO

Welcome back to a physical conference – after two years of pandemic, we are again able to hold a non-virtual conference. Corona has changed the world – there certainly will be more virtual interaction than before – witness VCOOP. Will physical conferences survive this seismic shift in ways to interact? Well, perhaps if the traditional conference format is adjusted to the new times. The ECOOP 2022 team has done a tremendous job of reigniting ECOOP – a huge thanx to them and their efforts – which have appeared to pay off, as both paper submission, attendance, and the number of workshops has increased. AITO will continue to explore new ways of adapting to the changing realities that scientific conferences face today – as spearheaded by the ECOOP 2022 organizers – we look forward to a really good conference with lots of great paper, personal interaction, excellent keynotes – including a Dahl-Nygaard Senior winner, Dan Ingalls. Enjoy the conference – and modern-day Berlin.

Eric Jul
AITO President

List of Authors

Pedro Adão (11)
Instituto Superior Técnico,
University of Lisbon, Portugal;
Instituto de Telecomunicações, Aveiro, Portugal

Giorgio Audrito (20)
University of Turin, Italy

Stephanie Balzer (22)
Carnegie Mellon University,
Pittsburgh, PA, USA

Heiko Becker (1)
MPI-SWS, Saarland Informatics Campus, (SIC),
Saarbrücken, Germany

John Bender (6)
Sandia National Laboratories,
Albuquerque, NM, USA

Oliver Bračevac (15)
Purdue University, West Lafayette, IN, USA

Frank Busse (9)
Imperial College London, UK

Julian Büning (9)
RWTH Aachen University, Germany

Cristian Cadar (9)
Imperial College London, UK

Roberto Casadei (20)
University of Bologna, Cesena, Italy

Madhurima Chakraborty (3)
University of California, Riverside, CA, USA

Ruo Fei Chen (22)
Independent Researcher, Leipzig, Germany

Ravi Chugh (16)
University of Chicago, IL, USA

Luca Ciccone (26)
University of Torino, Italy

Guillermina Cledou (27)
HASLab, INESC TEC, Porto, Portugal;
University of Minho, Braga, Portugal

Luke D'Alessandro (31)
Indiana University, Bloomington, IN, USA

Francesco Dagnino (26)
University of Genova, Italy

Ferruccio Damiani (20)
University of Turin, Italy

Eva Darulova (1, 13)
Uppsala University, Sweden

Kevin Deweese (31)
Cadence Design Systems, San Jose, CA, USA

Luc Edixhoven (27)
Open University of the Netherlands,
Heerlen, The Netherlands;
NWO-I, Centrum Wiskunde & Informatica,
Amsterdam, The Netherlands

Sebastian Erdweg (7)
JGU Mainz, Germany

Michael D. Ernst (10)
University of Washington, Seattle, WA, USA

Andong Fan (18)
Zhejiang University, Hangzhou, China

Sebastian Faust (35)
Technische Universität Darmstadt, Germany

Jesun Firoz (31)
Pacific Northwest National Laboratory,
Richland, WA, USA

Anthony Fox (1)
Arm Limited, Cambridge, UK

José Fragoso Santos (11)
Instituto Superior Técnico, University of Lisbon,
Portugal; INESC-ID Lisbon, Portugal

François Gauthier (29)
Oracle Labs, Brisbane, Australia

Lorenzo Gheri (8)
Imperial College London, UK

Mahdi Ghorbani (33)
University of Edinburgh, UK

Zachary Grannan (13)
University of British Columbia,
Vancouver, Canada

Dawu Gu (21)
Shanghai Jiao Tong University, China

Behnaz Hassanshahi (3, 29)
Oracle Labs, Brisbane, Australia

Dongjie He (30)
The University of New South Wales,
Sydney, Australia

Brian Hempel (16)
University of Chicago, IL, USA

Lukáš Holík (19)
FIT, Brno University of Technology,
Czech Republic

Xuejing Huang (18, 25)
The University of Hong Kong, China

Jules Jacobs (23)
Radboud University Nijmegen, The Netherlands

Sung-Shik Jongmans (27)
Open University of the Netherlands,
Heerlen, The Netherlands;
NWO-I, Centrum Wiskunde & Informatica,
Amsterdam, The Netherlands

Martin Kellogg (10)
University of Washington, Seattle, WA, USA

David Kretzler (35)
Technische Universität Darmstadt, Germany

Ramana Kumar (1)
DeepMind, London, UK

Nicolas Lagaillardie (4)
Department of Computing,
Imperial College London, UK

Ivan Lanese (8)
Focus Team, University of Bologna, Italy;
Focus Team, INRIA, Sophia Antipolis, France

Hehao Li (21)
Shanghai Jiao Tong University, China

Juanru Li (21)
Shanghai Jiao Tong University, China

Ben Liblit (14)
Amazon Web Services, Arlington, VA, USA

Kenan Liu (32)
SUNY Binghamton, NY, USA

Shuyang Liu (6)
University of California, Los Angeles, CA, USA

Xu Tony Liu (31)
University of Washington, Seattle, WA, USA

Yu David Liu (32)
SUNY Binghamton, NY, USA

Jingbo Lu (30)
The University of New South Wales,
Sydney, Australia

Andrew Lumsdaine (31)
University of Washington, Seattle, WA, USA;
Pacific Northwest National Laboratory,
Richland, WA, USA;
TileDB, Inc., Cambridge, MA, USA

Khaled Mahmoud (32)
SUNY Binghamton, NY, USA

Trong Nhan Mai (29)
Oracle Labs, Brisbane, Australia

Filipe Marques (11)
Instituto Superior Técnico,
University of Lisbon, Portugal;
INESC-ID Lisbon, Portugal

Daniel Marshall (5)
School of Computing, University of Kent,
Canterbury, UK

Scott McMillan (31)
Software Engineering Institute, Carnegie Mellon
University, Pittsburgh, PA, USA

Mira Mezini (35)
Technische Universität Darmstadt, Germany

Rajdeep Mukherjee (14)
Amazon Web Services, San Jose, CA, USA

Magnus O. Myreen (1)
Chalmers University of Technology,
Gothenburg, Sweden

Marcelo Navarro (34)
Computer Science Department (DCC),
University of Chile, Santiago, Chile

Rumyana Neykova (4)
Department of Computer Science,
Brunel University London, UK

Martin Nowack (9)
Imperial College London, UK

David Nowak (12)
Univ. Lille, CNRS, Centrale Lille, UMR 9189
CRIStAL, F-59000 Lille, France

Renzo Olivares (3)
University of California, Riverside, CA, USA

Bruno C. d. S. Oliveira (2, 18, 25)
The University of Hong Kong, China

Federico Olmedo (34)
Computer Science Department (DCC),
University of Chile, Santiago, Chile

Dominic Orchard (5)
School of Computing, University of Kent, Canterbury, UK;
Department of Computer Science and Technology, University of Cambridge, UK

André Pacak (7)
JGU Mainz, Germany

Luca Padovani (26)
University of Torino, Italy

Jens Palsberg (6)
University of California, Los Angeles, CA, USA

Petr Peringer (19)
FIT, Brno University of Technology, Czech Republic

Martin Plümicke (28)
Duale Hochschule Baden-Württemberg Stuttgart, Campus Horb, Germany

José Proença (27)
CISTER, ISEP, Polytechnic Institute of Porto, Portugal

Robert Rabe (1)
TU München, Germany

John Phillip Ratzloff (31)
SAS Institute, Cary, NC, USA

Baber Rehman (25)
The University of Hong Kong, China

David Richter (35)
Technische Universität Darmstadt, Germany

Adam Rogalewicz (19)
FIT, Brno University of Technology, Czech Republic

Tiark Rompf (15)
Purdue University, West Lafayette, IN, USA

Vlad Rusu (12)
Inria, Lille, France

Guido Salvaneschi (20, 35)
Universität St. Gallen, Switzerland

Nuno Santos (11)
Instituto Superior Técnico, University of Lisbon, Portugal; INESC-ID Lisbon, Portugal

Abhiroop Sarkar (17)
Chalmers University of Technology, Gothenburg, Sweden

Neil Sayers (8)
Imperial College London, UK;
Coveo Solutions Inc., Canada

Daniel Schemmel (9)
Imperial College London, UK

Max Schlüter (29)
Oracle Labs, Brisbane, Australia

Benjamin Selwyn-Smith (29)
Oracle Labs, Brisbane, Australia

Manuel Serrano (24)
Inria/UCA, Inria Sophia Méditerranée, 2004 route des Lucioles, Sophia Antipolis, France

Narges Shadab (10)
University of California, Riverside, CA, USA

Hesam Shahrokhi (33)
University of Edinburgh, UK

Amir Shaikhha (33)
University of Edinburgh, UK

Mary Sheeran (17)
Chalmers University of Technology, Gothenburg, Sweden

Manu Sridharan (3, 10)
University of California, Riverside, CA, USA

Andreas Stadelmeier (28)
Duale Hochschule Baden-Württemberg Stuttgart, Campus Horb, Germany

Alexander J. Summers (13)
University of British Columbia, Vancouver, Canada

Yaozhu Sun (18)
The University of Hong Kong, China

Bo Joel Svensson (17)
Chalmers University of Technology, Gothenburg, Sweden

Yong Kiam Tan (1)
Carnegie Mellon University, Pittsburgh, PA, USA

Zachary Tatlock (1)
University of Washington, Seattle, WA, USA

Peter Thiemann (28)
Institut für Informatik, Universität Freiburg, Germany

Bernardo Toninho (22)
NOVA LINCS, Nova University Lisbon, Portugal

Omer Tripp (14)
Amazon Web Services, San Jose, CA, USA

Emilio Tuosto (8)
Gran Sasso Science Institute, L'Aquila, Italy

Niki Vazou (13)
IMDEA Software Institute, Madrid, Spain

Mirko Viroli (20)
University of Bologna, Cesena, Italy

Tomáš Vojnar (19)
FIT, Brno University of Technology,
Czech Republic

Yizhuo Wang (21)
Shanghai Jiao Tong University, China

Guannan Wei (15)
Purdue University, West Lafayette, IN, USA

Pascal Weisenburger (35)
Universität St. Gallen, Switzerland

Micah Williams (29)
Oracle, Durham, NC, USA

Michael Wilson (14)
Amazon Web Services, Seattle, WA, USA

Anxhelo Xhebraj (15)
Purdue University, West Lafayette, IN, USA

Ningning Xie (25)
University of Cambridge, UK

Han Xu (18)
Peking University, Beijing, China

Jingling Xue (30)
The University of New South Wales,
Sydney, Australia

Joonhwan Yoo (32)
SUNY Binghamton, NY, USA

Nobuko Yoshida (4, 8)
Imperial College London, UK

Marcin Zalewski (31)
NVIDIA, Seattle, WA, USA

Yiwei Zhang (21)
Shanghai Jiao Tong University, China

Jinxu Zhao (2)
Department of Computer Science,
The University of Hong Kong, China

Florian Zuleger (19)
Faculty of Informatics, TU Wien, Austria

Veronika Šoková (19)
FIT, Brno University of Technology,
Czech Republic

Verified Compilation and Optimization of Floating-Point Programs in CakeML

Heiko Becker
MPI-SWS, Saarland Informatics Campus (SIC), Saarbrücken, Germany

Robert Rabe
TU München, Germany

Eva Darulova
Uppsala University, Sweden

Magnus O. Myreen
Chalmers University of Technology, Gothenburg, Sweden

Zachary Tatlock
University of Washington, Seattle, WA, USA

Ramana Kumar
DeepMind, London, UK

Yong Kiam Tan
Carnegie Mellon University, Pittsburgh, PA, USA

Anthony Fox
Arm Limited, Cambridge, UK

Abstract

Verified compilers such as CompCert and CakeML have become increasingly realistic over the last few years, but their support for floating-point arithmetic has thus far been limited. In particular, they lack the "fast-math-style" optimizations that unverified mainstream compilers perform. Supporting such optimizations in the setting of verified compilers is challenging because these optimizations, for the most part, *do not* preserve the IEEE-754 floating-point semantics. However, IEEE-754 floating-point numbers are finite approximations of the real numbers, and we argue that any compiler correctness result for fast-math optimizations should appeal to a real-valued semantics rather than the rigid IEEE-754 floating-point numbers.

This paper presents RealCake, an extension of CakeML that achieves end-to-end correctness results for fast-math-style optimized compilation of floating-point arithmetic. This result is achieved by giving CakeML a flexible floating-point semantics and integrating an external proof-producing accuracy analysis. RealCake's end-to-end theorems relate the I/O behavior of the original source program under real-number semantics to the observable I/O behavior of the compiler generated and fast-math-optimized machine code.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Compilers; Software and its engineering → Software performance

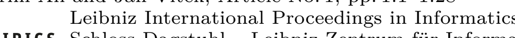
Keywords and phrases compiler verification, compiler optimization, floating-point arithmetic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.1

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.10`

Funding *Eva Darulova*: Part of this work was done while the author was at MPI-SWS, Germany.
Magnus O. Myreen: Supported by the Swedish Foundation for Strategic Research.
Zachary Tatlock: Supported in part by the U.S. Department of Energy under Award Number DE-SC0022081 (ComPort), and by the National Science Foundation under Grant No. 1749570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DOE or NSF.

© Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 1; pp. 1:1–1:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Verified compilers guarantee that the executable code they generate will only exhibit behaviors allowed by the semantics of the input program. Establishing such guarantees is challenging [49], especially if the compiler is to perform sophisticated optimizations. Adding new classes of optimizations requires significant design and proof engineering effort. Despite tremendous progress, state-of-the-art verified compilers like CakeML [55] for ML and CompCert [32] for C remain only moderately optimizing. While CakeML and CompCert support classic optimizations like common subexpression and dead code elimination, their compilation and optimization of floating-point programs is very limited: CompCert performs only a few conservative optimizations and, prior to this work, CakeML did not support floating-point arithmetic at all.

The limited support of floating-point programs in verified compilers is in stark contrast to mainstream compiler frameworks like GCC [19] and LLVM [4]. Both of these compiler frameworks support aggressive floating-point optimization via their `fast-math` flags [20, 57, 33, 7]. Fast-math optimizations include reassociating arithmetic, *e.g.*, rewriting $x \times (x \times (x \times x)) \rightarrow (x \times x) \times (x \times x)$ to enable common subexpression elimination; fused-multiply-add (FMA) introduction, *i.e.*, rewriting $x \times y + z \rightarrow \mathtt{fma}(x, y, z)$ for strength reduction and to avoid intermediate rounding; as well as branch folding and dead code elimination by assuming special floating-point values like Not-a-Number (NaN) do not arise.

While such optimizations are sound under real-number semantics, they generally do *not* preserve IEEE-754 behavior [26], *e.g.*, because floating-point arithmetic is non-associative due to the inherent rounding at every intermediate operation. In part, this is why verified compilers up until now have not supported fast-math optimizations: CompCert strictly preserves IEEE-754 semantics [10], under which such optimizations are disallowed.

However, for many applications strict preservation of IEEE-754 semantics is overly constraining and artificial, preventing useful performance optimizations. Numerical applications are typically *designed* implicitly assuming real-number arithmetic and are only later *implemented* in floating-point arithmetic.[1]

The Icing language [5] proposes a more relaxed, non-deterministic semantics for floating-point expressions that allows a limited set of fast-math-style optimizations to be applied. Icing comes with a proof-of-concept optimizer, whose formal correctness proof shows that the optimization result is one of those modeled by the semantics of the initial expression.

While Icing showed what it means to allow fast-math-style optimizations in a verified compiler, Icing did not go as far as bounding the accuracy of the resulting, fast-math-optimized code. Icing's correctness theorems describe only the optimizations that a verified compiler can apply to a floating-point expression, but not their effect on overall program behavior. Hence the Icing optimizer cannot bound changes in the accuracy of the optimized floating-point expression with respect to the real-valued semantics of the unoptimized expression.

We argue that a verified compiler must provide accuracy guarantees to reasonably support fast-math-style optimizations. Applications in domains such as signal processing [12], embedded controllers [38], and neural networks [23], which could be optimized with fast-math-style optimizations, are designed to operate in noisy environments and can thus tolerate a certain amount of floating-point roundoff error by design, however, this noise has to be *bounded*. For example, a real-number version of an embedded controller is typically proven correct (i.e. stable) with respect to *bounded* implementation noise [3]. At the same time, performance is important and so developers are often indifferent to fine-grained floating-point implementation decisions.

[1] Error-free computation with rational or constructive real [8] libraries is often prohibitively expensive.

To support such potentially safety-critical applications in a verified compiler, we introduce a local, more flexible notion of correctness which we call *error refinement*: a floating-point kernel within an application may be optimized (potentially changing its IEEE-754 behavior) as long as its results remain within a user-specified error bound relative to the implicit real-number semantics.

We formalize error refinement inside the CakeML compiler to support fast-math optimizations *end-to-end*. Our extension, which we call *RealCake*, carries source-level guarantees down to fast-math-optimized executable machine code. That is, our final correctness theorem shows that running the *machine code* for a fast-math-optimized floating-point program under strict IEEE-754 semantics produces a result that is within a programmer-provided error bound w.r.t. the *unoptimized program* evaluated under real-number semantics. While our extension is done in the context of the CakeML compiler, we expect it to carry over to other verified compilers like CompCert as well.

Our first key technical contribution is a relaxed floating-point semantics that allows both fast-math-style optimizations as well as *backward simulation* soundness proofs (as CakeML's semantics requires determinism). RealCake's relaxed semantics preserves the core ideas of the existing Icing semantics [5] and models nondeterministic application of an arbitrary number of fast-math rewrites, just as Icing does. Icing provides only a loose coupling with CakeML via a simulation between the deterministic optimized expression and a CakeML source program. Unlike Icing, RealCake's semantics is designed to be more tightly integrated with the CakeML source semantics. This new integration is necessary to prove end-to-end error refinement that relates *unoptimized real-valued* CakeML programs and *optimized floating-point machine code*. RealCake's design furthermore supports function calls, I/O and memory beyond (Icing supported) floating-point expressions and can thus prove error refinement for complete applications.

The second technical contribution is to realize error refinement with *translation validation* [45, 51] using an interface to an existing proof-producing roundoff error bound analysis [6]. RealCake automatically composes the error bound proofs with its optimizer's correctness theorems to support fast-math optimizations with semantic and accuracy guarantees within a verified compiler for the first time.

RealCake is primarily designed to support *numerical kernels*, straight-line code such as those found in (safety-critical) embedded controllers or sensor-processing applications.[2] Often such kernels are evaluated in a control loop or process sensor inputs repeatedly. For such programs, both correctness as well as performance are important, and an analysis of the straight-line code is sufficient: the correctness (stability) of the overall programs (and loops) can be shown with, *e.g.*, control-theoretic techniques that rely on the straight-line loop body's errors being bounded [3, 37]. We do not address some of the orthogonal challenges in bounding the floating-point roundoff error for programs with loops and conditional statements, which remain open research problems [15]; state-of-the-art proof-producing error bound analyses only robustly support straight-line numerical kernels [54, 41, 6]. A key aspect of RealCake's design is the loose coupling between the compiler and error analysis. This loose coupling leads to a clean separation of concerns, which we hope will allow us to switch to more general error analysis methods when such are discovered.

We evaluated RealCake by optimizing all kernels from the standard floating-point arithmetic benchmark suite FPBench [14] (Section 7) which can be expressed as input to RealCake, for a total of 51 kernels. During our evaluation we found that CakeML was missing a general

[2] RealCake nevertheless proves error refinement for whole programs, including I/O (Section 7).

```
(* target error bound: 2^-5,
   precondition P:
   0.0 ≤ x1 ≤ 5.0 ∧ −20.0 ≤ x2 ≤ 5.0 *)
fun jetEngine(x1:double, x2:double):double =
  opt: (let
  val t = (((3.0 * x1) * x1) + (2.0 * x2)) - x1
  val t2 = (((3.0 * x1) * x1) - (2.0 * x2))
    - x1
  val d = (x1 * x1) + 1.0
  val s = t / d
  val s2 = t2 / d
  in
  x1 + ((((((((2.0 * x1) * s) * (s - 3.0)) +
  ((x1 * x1) * ((4.0 * s) - 6.0))) * d) +
  (((3.0 * x1) * x1) * s)) +
  ((x1 * x1) * x1)) + x1) + (3.0 * s2))
  end)
```

(a) Unoptimized floating-point kernel.

```
(* guaranteed error bound: 2^-5,
   precondition P:
   0.0 ≤ x1 ≤ 5.0 ∧ −20.0 ≤ x2 ≤ 5.0 *)
fun jetEngine(x1:double, x2:double):double =
  noopt: (let
  val t = fma((x1+x1)+x1, x1, (x2 + x2) - x1)
  val t2 = fma((x1+x1)+x1, x1,
    fma(-2.0, x2, -x1))
  val d = fma(x1, x1, 1.0)
  val s = t / d
  val s2 = t2 / d
  in
  x1 + fma(x1 * d, fma((s - 3.0) + (s - 3.0),
    s, x1 * fma(4.0, s, -6.0)),
    fma(x1 * x1, ((s + s) + s) + x1,
    x1 + ((s2 + s2) + s2)))
  end)
```

(b) Optimized floating-point kernel.

```
fun main () = let
  val args = Commandline.arguments ()
  val a = Double.fromString (List.nth args 1)
  val b = Double.fromString (List.nth args 2)
  val r = jetEngine (a, b)
  in
   TextIO.print (Double.toString r)
  end
```

(c) Main function.

Figure 1 Example unoptimized and optimized CakeML floating-point kernels, and a stand-in main function. The `opt:` annotation (lines 5) allows developers to selectively apply optimizations. Here, we choose to optimize the entire kernel, but a user may place only part of a program under `opt:` and the rest will be compiled preserving IEEE-754 semantics.

optimization that is particularly effective for floating-point programs: global constant lifting. RealCake achieves a (geometric) mean performance improvement for fast-math optimizations of 3% and a maximum improvement of 16% on top of improvements from constant lifting with respect to the unoptimized FPBench kernels. Our additional constant lifting optimization achieves a geometric mean performance improvement of 83% across all benchmarks with speedups of up-to 97%. For all optimized kernels, RealCake formally guarantees that the roundoff error remains within a user-specified bound.

Contributions

To summarize, this paper makes the following contributions:

- the concept of error refinement and its formalization within the CakeML verified compiler (Section 2);
- an extension of CakeML with strict, IEEE-754 semantics preserving, floating-point arithmetic as well as a relaxed non-deterministic floating-point semantics (Section 4);
- a fast-math optimizer that is effective in improving the performance of floating-point programs (Section 5);
- automated proof tools that soundly bound roundoff errors of (optimized or unoptimized) kernels w.r.t. our new real-number semantics for CakeML (Section 6).

The RealCake development is publicly available.

2 Overview

We start by demonstrating at a high-level how RealCake works using an example, before giving an overview of the RealCake toolchain and our major design decisions.

2.1 Example

Figure 1a shows `jetEngine`, a straight-line nonlinear embedded controller [3] adapted to CakeML syntax. This controller has been proven to be safe for the dynamical system of a jet engine compressor. That is, it has been proven that the two state variables (x_1, x_2) will remain within the bounds given by P (on line 2), that the system will always steer the state variables towards the equilibrium point (0.0), and that the systems thus remains stable. The stability proof assumes the control expression to be real-valued, but accounts for a certain amount of error, including measurement and implementation errors, and hence the controller is stable as long as the errors remain below this bound. For the purpose of this paper, we choose 2^{-5} as the bound on the roundoff error due to a floating-point implementation. However, in addition to stability, performance is also an important concern when designing controllers for resource-constrained embedded systems. To summarize, an embedded developer designs a controller, such as the `jetEngine`, assuming real-valued arithmetic together with an error bound, and requires that the executed finite-precision code A) correctly implements the control expression, B) is as efficient as possible, and C) meets the error bound.

Such a kernel may be part of a safety critical system so we would like to compile it to an executable using a verified compiler. Unfortunately, no verified compiler today meets the requirements listed above: while CompCert [10] does support floating-point arithmetic, and so ensures A), it does not optimize floating-point programs and cannot provide roundoff error bounds. Prior to our work, CakeML did not even support floating-point arithmetic.

RealCake, our extension of the CakeML compiler, closes this gap. RealCake automatically optimizes the input kernel into the optimized version shown in Figure 1b, compiles it down to machine code, and proves the end-to-end correctness theorem shown in Figure 2 that captures both "traditional" compiler correctness as well as accuracy guarantees.

For this example, RealCake prepares the program for optimization by replacing floating-point subtraction by addition of the inverse ($(4.0 \times s) - 6.0 \to (4.0 \times s) + (-1 \times 6.0)$), and during optimization, RealCake replaces multiplications by additions ($2.0 \times x1 \to x1 + x1$), and introduces FMA instructions ($x1 \times x1 + 1.0 \to \texttt{fma}(x1, x1, 1.0)$) that go beyond IEEE-754 semantics. For this example, RealCake compiles the optimized floating-point kernel (Figure 1b) together with a simple stand-in main function (Figure 1c) into a verified binary. On a Raspberry Pi v3, RealCake improves the performance of our example kernel by 95%. This performance improvement comes from both floating-point specific optimizations, as well as global constant lifting that is not specific but particularly effective for floating-points and that CakeML did not support before (Section 7). Such a speedup is important for repeatedly run code such as our embedded controller.

RealCake automatically proves the end-to-end correctness theorem that we formally state in Figure 2. At a high-level, Theorem 1 relates the behavior of the optimized program with the behavior of the real-number semantics of the initial, unoptimized program on the domain specified by precondition P.[3]

[3] The precondition is important, since roundoff errors directly depend on the ranges of (intermediate) values.

▶ **Theorem 1** (*jetEngine* – Whole program correctness).

$\texttt{jetEngineInputsInPrecond}\ (s_1, s_2)\ (w_1, w_2)\ \wedge\ \texttt{environmentOk}\ ([\mathit{jetEngine};\ s_1;\ s_2], \mathit{fs})\ \Rightarrow$
$\exists\, w\ r.$
$\quad \texttt{CakeMLevaluatesAndPrints}\ (\mathit{jetEngineCode}, s_1, s_2, \mathit{fs})\ (\texttt{toString}\ w)\ \wedge$
$\quad \texttt{initialFPcodeReturns}\ \mathit{jetEngineUnopt}\ (w_1, w_2)\ w\ \wedge$
$\quad \texttt{realSemanticsReturns}\ \mathit{jetEngineUnopt}\ (w_1, w_2)\ r\ \wedge\ \texttt{abs}\ (\texttt{fpToReal}\ w\ -\ r)\ \leq\ 2^{-5}$

Figure 2 RealCake-proven specification theorem for the `jetEngine` kernel. Here, *jetEngineCode* refers to the overall program consisting of the *jetEngine* kernel, the *main* function from Figure 1c, and the glue-code for I/O, *jetEngine* is the name of the produced binary, and *jetEngineUnopt* is the kernel from Figure 1a.

Formally, the theorem states that, if the kernel is run on two arbitrary input strings s_1 and s_2, representing the double word inputs w_1 and w_2 respectively, and the double words satisfy precondition P from Figure 1a (assumption `jetEngineInputsInPrecond`), and if the machine code for the `jetEngine` kernel is run with three command line arguments (the name of the binary, and s_1 and s_2) in an environment with filesystem `fs` (assumption `environmentOk`), then there exists a double-precision floating-point word w such that

(a) running the optimized kernel with the command line arguments prints the word w on stdout[4]

(b) w is a result of running the unoptimized `jetEngine` kernel on (w_1, w_2) with optimizations applied by our relaxed semantics, and

(c) running the *initial unoptimized* `jetEngine` kernel under real-number semantics on (w_1, w_2) returns a real number r such that $|w - r| \leq 2^{-5}$, where 2^{-5} is the user-given error bound.

2.2 Overview of CakeML

RealCake extends the CakeML compiler toolchain [55], built around a verified compiler for (a dialect of) Standard ML (SML). CakeML compiles programs written in SML to x86, ARMv7, ARMv8, MIPS, RISC-V and Silver [35] machine code and is implemented completely in the HOL4 theorem prover [53]. Our work mainly focuses on the compiler part of the CakeML ecosystem.

The behavior of a program written in the CakeML dialect of SML is defined in the CakeML source semantics. This semantics is implemented as a deterministic function in the HOL4 theorem prover, in the style of functional big-step semantics [43]. CakeML programs are turned into machine code using the in-logic compiler, with compiler passes going through various intermediate languages.

The CakeML compiler's correctness theorem states that the compiler preserves observable behaviors of the input program, modulo out-of-memory errors that can occur in the generated machine code [21].

[4] We chose printing to standard output as one option for implementing I/O behavior to show how the error bound proof can be related to I/O behavior. In a real-world setting this could be replaced by other I/O functionality.

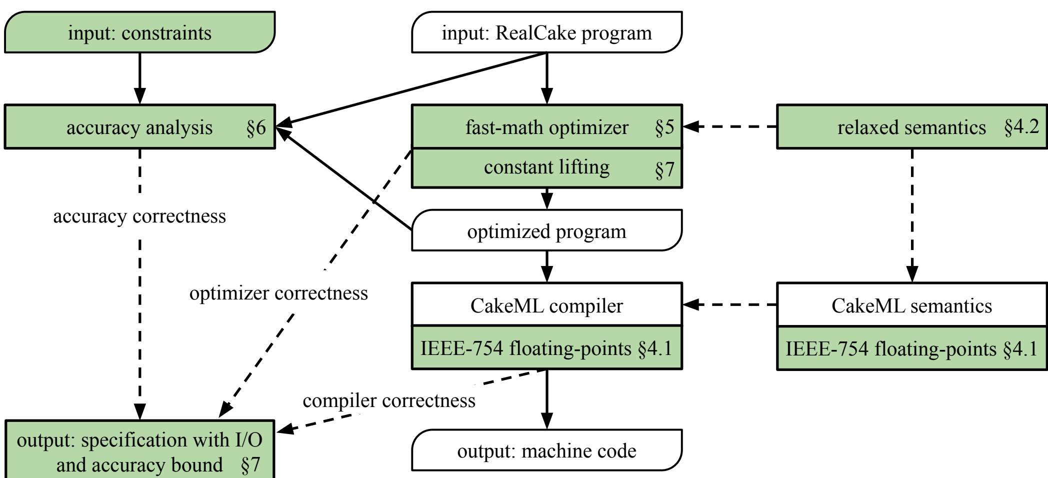


Figure 3 Overview of the RealCake toolchain. Boxes with white background are part of the original CakeML toolchain, our RealCake extensions are marked with a green background, dashed lines indicate proof dependencies, solid lines indicate output flows.

2.3 Overview of RealCake

Figure 3 illustrates the RealCake toolchain, with the extensions over CakeML marked in green. The RealCake toolchain takes as inputs a program and constraints similar to those in Figure 1a. In a first step, the fast-math optimizer is run on each floating-point kernel, optimizing it with respect to RealCake's relaxed floating-point semantics, as well as lifting constants. As a result we obtain an optimized floating-point kernel, and a proof relating executions of the optimized kernel back to its unoptimized version. Next, the input constraints, and the optimized kernel are run through our accuracy analysis pipeline (left part of Figure 3). We have proven once and for all that if the analysis succeeds, the roundoff error of the optimized floating-point kernel with respect to a real-number semantics of its unoptimized version is below the user specified error bound. This requires non-trivially combining properties of the fast-math optimizer with a simulation proof relating results of the roundoff error analysis to CakeML floating-point programs. Finally, the CakeML compiler compiles the optimized kernel into machine code that can be run on x86-64 and ARMv7 platforms.[5] RealCake automatically combines optimizer correctness with the correctness of the accuracy analysis and the CakeML compiler correctness theorem to prove a theorem about the *I/O behavior* and the *accuracy* of the machine code with respect to the real-number semantics of the unoptimized, initial kernel.

One of our key insights is to apply the fast-math optimizations that require reasoning about nondeterministic semantics early; a nondeterministic semantics can be integrated more easily with a deterministic verified compiler by resolving the nondeterminism before the code enters the compiler itself. We formally prove the correctness of the fast-math optimizer: if the optimizer turns kernel p_1 into kernel p_2, and evaluating p_2 returns the floating-point word w, then the relaxed floating-point semantics can evaluate and optimize p_1 such that it returns w.

[5] At the time of writing, only the underlying ISA models for x86-64 and ARMv7 support floating-point arithmetic in CakeML.

2.4 Error Refinement

Optimization correctness alone effectively captures only the machine's point of view, and ignores the programmer's (implicit) real-valued source semantics. To relate the real-number, unoptimized program with its fast-math-optimized version requires both proving the correctness of our optimizer (*i.e.* showing that the behavior of the source semantics is preserved), as well as establishing accuracy guarantees using roundoff errors. Any fast-math optimization will necessarily change the rounding and thus the result value of the floating-point kernel, ruling out alternative bit-wise comparisons. While the programmer will be indifferent to how exactly the floating-point code is compiled and will accept some roundoff error – or she would not have chosen finite-precision arithmetic – this roundoff error should not be unduly large and make the computed results useless.[6] We argue that a correctness theorem of verified fast-math floating-point compilation thus needs to capture this error refinement.

To this end, RealCake automatically infers verified accuracy bounds via a verified translation from CakeML source to the proof-producing formally verified static analysis tool FloVer [6]. We combine a simulation proof relating the floating-point semantics of FloVer and CakeML with a proof that all optimizations done by our fast-math optimizer are real-valued identities. RealCake then lifts the roundoff error bound to the complete program and combines it with the general compilation correctness proofs to automatically show the end-to-end correctness theorem for our example (Figure 2). This makes RealCake the first verified compiler for floating-point arithmetic that proves a whole-program specification relating the I/O behavior of optimized floating-point machine code to the real-number semantics of the unoptimized initial program.

We choose to integrate the roundoff error analysis only loosely with CakeML. This gives us a flexible compiler infrastructure that allows us to prove roundoff error bounds on optimized as well as unoptimized floating-point kernels, or to greedily optimize kernels without necessarily proving roundoff error bounds (but still obtaining compiler correctness guarantees). By not tightly integrating the roundoff error analysis into CakeML, we have the option to relatively easily replace FloVer with an extension or another tool in the future.

RealCake's end-to-end correctness theorem only relies on error bounds proven independently for each straight-line kernel instead of a global kernel error bound. Our focus on straight-line kernels is inherited from the current capabilities of verified floating-point error analyses (see Subsection 3.2 for a more detailed discussion), but can be easily lifted with advances in this area. Per-kernel error analysis, on the other hand, is crucial to maintaining compiler modularity: it is not (nor should it be) the compiler's responsibility to ensure that a program is globally numerically stable – that is a job for the algorithm designer. Rather, the compiler compiles *and optimizes* a program and, in the case of fast-math floating-point optimizations, ensures that it has preserved sufficient (user-provided) accuracy bounds with respect to a specification over real-number semantics. This can be checked locally.

Similarly, the goal of the accuracy analysis is not necessarily to improve the accuracy of a given kernel, even though introducing FMAs will generally have this effect, but rather to ensure that the compiler has not introduced unacceptable numerical instability by accident.

Overall, a key challenge of RealCake is proof engineering. RealCake combines a verified roundoff error analysis with the deterministic CakeML compiler and a non-deterministic semantics that supports floating-point optimizations. Specifically, the main proof engineering

[6] There are programs, such as compensated sum algorithms [25], that explicitly rely on the exact floating-point semantics; such code would not be subject to fast-math-style optimizations and thus not written under an `opt:` scope.

challenge is getting the different tools to "cooperate". CakeML's source semantics is an integral part of the CakeML ecosystem. Therefore, our integration of the relaxed floating-point semantics must make sure to not break any existing invariants. Further, the semantics of the external roundoff error analysis and the semantics of CakeML source programs must be compatible such that analysis results can be transformed into CakeML source properties. Finally, all of this has to happen while making sure that RealCake optimizes floating-point programs with a non-deterministic relaxed floating-point semantics.

3 Background

In this section, we review some necessary background on IEEE-754 floating-point arithmetic, how to analyze the roundoff error of IEEE-754 floating-point programs, and how the Icing semantics allows to support fast-math-style optimizations in a verified compiler context.

3.1 IEEE-754 Floating-Point Arithmetic

The IEEE-754 standard [26] defines the representation, special values, arithmetic operations, and rounding modes of floating-point arithmetic. A floating-point number x is represented as a triple (s, m, e) that defines $x = (-1)^s \times m \times 2^e$ where s is the sign bit, m the so-called significand and e the exponent. Most commonly used formats are binary single `float` and `double` precision that use 23 and 52 bits for the significand and 8 and 11 bits for the exponent, respectively. If the exponent of a number is 0, it is called a subnormal number, all other valid numerical values are called normal numbers. IEEE-754 additionally defines the special values *Infinity* and *NaN* that represent exceptional results. The standard further specifies that arithmetic operations (*e.g.*, $+, -, \times, /$) are rounded correctly, i.e. the result is as if the computation was performed in infinite precision and then rounded. The standard defines five rounding modes, of which we assume and support the most commonly used: rounding to nearest, ties to even. A further consequence of the finite precision and rounding is that floating-point arithmetic does not satisfy common real-valued identities such as associativity and distributivity. Hence, reordering a computation may lead to different results (and roundoff errors), even though the expression is equivalent under the reals.

3.2 Analysis of Rounding Errors

There exist a number of analysis tools that bound absolute roundoff errors for floating-point kernels and that provide formally verified error bounds: Precisa [56], FPTaylor [54], real2Float [36], Gappa [17], and FloVer [6]. They either use a global optimization analysis approach, or a forward dataflow analysis using an interval abstract domain to compute absolute roundoff errors:

$$\max_{x \in P(x)} |f(x) - \tilde{f}(x)| \tag{1}$$

where f is the real-number expression, $\tilde{f}$ its floating-point counterpart and $P(x)$ is the precondition that constrains the input variables x. A precondition providing lower and upper bounds on the inputs is necessary to obtain interesting, non-infinite roundoff error bounds. Computing relative errors $|f(x) - \tilde{f}(x)|/|f(x)|$ is not well-defined when the denominator is zero and is thus not suitable for a general error analysis. For our purpose of checking that compilation has not introduced large numerical instabilities, any of the above mentioned tools is in principle suitable. We choose FloVer, because it is conveniently implemented in HOL4.

Despite the abundance of analysis tools, bounding finite-precision roundoff errors remains a challenging and active research area. We thus choose to focus on verifying absolute roundoff errors w.r.t. a real-number specification for straight-line arithmetic kernels, which is currently well supported. To the best of our knowledge, all available verified roundoff error analysis tools relate roundoff errors to idealized real-number semantics. Support for conditionals and loops [41] is currently severely limited. The challenge with loops is that roundoff errors in general grow in every loop iteration and thus computed fixpoints necessarily become infinite. We thus focus on absolute error accuracy analysis of straight-line arithmetic kernels, i.e. binary arithmetic operations $(+, -, \times, /)$, unary $-$, `fma` operations and let-bindings.

FloVer is a verified certificate checker for finite-precision roundoff errors that is meant to validate results of external, unverified, floating-point analysis tools. Given a certificate, encoding the result of the external analysis tool, FloVer verifies the bounds encoded in the certificate by computing the bounds using a dataflow analysis in logic. In this work in RealCake, we extend FloVer with an unverified function that computes a roundoff error certificate, which we then send through the checking pipeline.

FloVer abstracts floating-point arithmetic operations by:

$$(x \circ_{fl} y) = (x \circ y)(1 + e) \quad |e| \leq \varepsilon \tag{2}$$

where $\circ \in \{+, -, \times, /\}$ and ε is the machine epsilon. FloVer uses interval arithmetic [40] to propagate intermediate bounds on x and y, on which the magnitude of absolute errors and error propagation depends, and equally uses interval arithmetic to propagate worst-case error bounds through the arithmetic expression. For our evaluation, we have added support for `sqrt` operations to FloVer which was previously unsupported.

FloVer's soundness theorem states that a successful run of FloVer implies that the encoded floating-point kernel can be run under IEEE-754 floating-point semantics, and that the given error bound is a sound upper bound on the worst-case absolute error between the floating-point execution and the idealized real-number semantics. Error bounds proven by FloVer in RealCake are valid for normal and subnormal floating point numbers. To make this possible we extended FloVer with support for subnormal floating-point numbers, and have proven the corresponding HOL4 theorems.

3.3 Icing Floating-Point Semantics

Icing [5] was proposed as the first semantics to support fast-math-style optimizations in a verified compiler, going beyond IEEE-754 floating-point semantics. To support fast-math-style optimizations, Icing relies on three core ideas: fine-grained control, giving the programmer full control over which part of the program is optimized; value trees, which are a lazy representation of floating-point values; and nondeterministic floating-point evaluation, applying optimizations while evaluating. We review the design rationale behind each of these points.

Fine-Grained Control In unverified compilers fast-math optimizations are globally switched on or off. For a verified compiler this is unsatisfactory, as some code can be heavily optimized, while some of it might need to be compiled under strict IEEE-754 semantics. Icing solved this issue by introducing fine-grained control over which part of a program is optimized by annotating it with `opt:`, if optimizations should be applied, and `noopt:`, if optimizations should not be applied.

Value Trees In Icing, floating-point values are not represented by 64-bit words. Instead, Icing uses a lazy datatype called value trees. A value tree is a tree with constants as leaf nodes, and operators as intermediate nodes. During evaluation, expression variables

Figure 4 Value trees for the unoptimized example expression (left) and its optimized version with an FMA instruction (right).

are replaced by a value tree loaded from an execution environment. When evaluating a floating-point comparison, value trees are eagerly compressed into floating-point words. Value trees are a perfect fit for encoding syntactic information about the evaluated expression in the Icing semantics.

Nondeterministic Semantics To handle fast-math-style optimizations in the semantics, Icing adds a set of allowed optimizations to the semantics, and the semantics can nondeterministically optimize by applying a subset of the allowed optimizations to value trees.
Icing's lazy value trees allow the semantics to alter the structure of a floating-point value after it has been evaluated. This is key to modelling the floating-point optimizations.

Example

To illustrate how the Icing semantics works and how optimizations are applied we give a simple example. We will optimize the floating-point expression `opt:(x * 3.5 + 9.1)` in the Icing semantics.

If we evaluate the initial, unoptimized expression (`opt:(x * 3.5 + 9.1)`) in Icing semantics, which nondeterministically optimizes by introducing FMA instructions, the semantics first computes the value tree at the left-hand side of Figure 4. Because of the nondeterminism, the semantics can now either keep the value tree as is, or introduce an FMA, replacing the value tree by the one on the right-hand side of Figure 4. We explain next how Icing establishes a relation between nondeterministic optimizations and evaluation of optimized expressions to prove correctness of optimizers.

Correctness Proofs

The original Icing paper presents three different optimizers. Here, we focus on the so-called *greedy optimizer*, and the *IEEE-754-translator*. For a list of optimizations, the greedy optimizer greedily applies them to a program wherever possible. The IEEE-754-translator rules out further optimizations by replacing all `opt:` optimization annotations by `noopt:`.

Correctness of the IEEE-754-translator proves that program evaluation is deterministic after applying the IEEE-754-translator, as no optimizations can be applied syntactically or semantically. The main correctness theorem for the greedy optimizer states: Suppose the greedy optimizer is run with optimizations o on floating-point program f and returns program g. If evaluating g without any optimizations under Icing semantics gives value tree v, then one of the results of evaluating f nondeterministically under Icing semantics with optimizations o enabled is also value tree v. In general we call such a proof a *backwards simulation*, as it relates the result obtained from evaluating the optimized program back to an evaluation of the initial program with the applied optimizations added to the semantics.

While being an important first step to support fast-math-style optimizations in a verified compiler, Icing is not able to prove accuracy guarantees, which are required to prove end-to-end error refinement. Even though the original Icing paper proposes to include roundoff

error bounds in future work ([5], Theorem 1), we found that these guarantees could not be translated into guarantees for CakeML programs, as the backwards simulation only allows transferring information from CakeML programs to Icing expressions, but not vice versa. This motivates our approach of tightly integrating a relaxed floating-point semantics with CakeML source semantics. It allows to establish accuracy guarantees and carry them down to machine code generated by the compiler.

4 RealCake's Semantics

Our overall goal for RealCake is to optimize and compile floating-point kernels, establishing verified end-to-end correctness and accuracy guarantees. In this section, we lay the foundations for this work by extending the CakeML compiler with three different semantics: strict IEEE-754 preserving floating-point arithmetic, relaxed floating-point semantics going beyond IEEE-754, and a real-number semantics as a ground truth for bounding errors.

4.1 Extending CakeML with IEEE-754 Floating-Point Arithmetic

Prior to this paper, CakeML did not have support for floating-point arithmetic. In a first step, we add strictly IEEE-754 compliant floating-point arithmetic to CakeML. This part of the work did not require any deep new insights; we briefly review the supported operations.

The CakeML source language already had support for 64-bit machine words and we used these to hold IEEE-754 double values, but added new primitive operations for 64-bit words (single precision floats are currently not supported). The new source-level primitive operations are floating-point addition, subtraction, multiplication, division; multiply-and-add, negation, square root, absolute value, and the usual floating-point comparisons. The semantics was defined using an existing formalization of IEEE-754 by Harrison [24], which includes NaNs and Infinities.[7]

The bulk of the compiler required only simple changes since most intermediate languages compile the strict IEEE-754 operations to their very similar counterparts in the next intermediate language. The only internal part that required a bit more effort is the point where the data abstraction is implemented, i.e. where all data becomes concrete. At this point we had to wrap every primitive floating-point operation with code that unboxes and then boxes the double values. The same code is also responsible for loading and storing to the architecture-specific floating-point registers. Our IEEE-754-preserving compilation ensures that evaluation order is preserved.

At the time of writing, the CakeML compiler has six target languages. We have added floating-point support to two of them: the x86-64 and ARMv7 targets. The ARMv7 model that we use already included IEEE-754 floating-point support based on the same standard formalization that the CakeML semantics uses. For x86-64, we extended the model of the x86-64 instruction set architecture to include a minimal collection of IEEE-754 floating-point instructions required by the compiler. As the underlying L3 model [18] of the x86-64 instruction set architecture does not support FMA instructions, only the ARMv7 backend currently supports code generation for kernels with FMA instructions.

[7] The fragment of IEEE-754 that we include in CakeML has not changed between standard revisions.

```
evaluate st env [App op es] =
case evaluate st env es of
(st', Rerr v) => (st', Rerr v)
(st', Rval vs) =>
  case do_app (st'.refs, st'.ffi) op vs of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) =>
    (updateState st' refs ffi, list_result r)
```

(a) Standard operator evaluation.

```
evaluate st env [FpOptimise ann e] =
case evaluate (updateOptFlag st ann) env [e] of
(st', Rerr e) => (resetOptFlag st' st, Rerr e)
| (st', Rval vs) =>
  (resetOptFlag st' st, Rval (addAnnot ann vs))
```

(b) Optimization scope evaluation.

```
evaluate st env [App op es] =
case evaluate st env es of
(st', Rerr v) => (st', Rerr v)
| (st', Rval vs) =>
  case do_app (st'.refs, st'.ffi) op vs of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) =>
  let (st', r_opt) = optimizeIfOk st' r
      fp_res = if isFpBool op then toBool r_opt
               else r_opt
  in
  (updateState st' refs ffi, list_result r)
```

(c) Relaxed floating-point evaluation.

```
evaluate st env [App op es] =
case evaluate st env es of
(st', Rerr v) => (st', Rerr v)
| (st', Rval vs) =>
if ¬realsAllowed st'.fp_state then
  (advanceOracle st',
    Rerr (Rabort Rtype_error))
else
  case do_app (st'.refs, st'.ffi) op vs of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) =>
    (updateState st' refs ffi, list_result r)
```

(d) Real-valued operator evaluation.

Figure 5 HOL4 definitions of operator evaluation in CakeML source for (a) the simple case, (b) relaxed floating-point operations, (c) optimization scopes, and (d) real-number operations. In (c) and (d) difference to Figure 5a is highlighted in bold.

4.2 RealCake's Relaxed Floating-Point Semantics

Next, we present RealCake's relaxed floating-point semantics. Similar to Icing, the relaxed floating-point semantics applies optimizations during evaluation. In CakeML, we call the process of applying optimizations to floating-point kernels during evaluation *semantic optimization*. Before going into the details of how evaluation is implemented in the relaxed semantics, we briefly review some necessary details of CakeML's source semantics.

CakeML Source Semantics

The CakeML source semantics is implemented in the style of functional big-step semantics [43]. As such, CakeML source semantics (`evaluate`) is a pure, deterministic function in the HOL4 theorem prover. `evaluate` st_1 env e = (st_2, r) means that evaluating the CakeML source expression e under environment env and global state st_1 results in global state st_2 and ends with result r. If evaluation succeeded, r is a value, otherwise r is an error. Global state st_1 and st_2 model the state of the foreign-function-interface (FFI), as well as global references. In CakeML source semantics, the FFI models interactions with the outside world, *e.g.* I/O.

We explain the case for operator evaluation in more detail, as we will extend it with relaxed floating-point operations later. The definition of operator evaluation in CakeML is given in Figure 5a. In CakeML source, an operator application is written as `App` *op es*, denoting that operator *op* is applied to the list of expressions *es*.

First, `evaluate` is run on the argument list. If evaluation of the argument list fails with an error and a new state, the error and the state are returned. If evaluation succeeds returning values *vs*, function `do_app` applies operator *op* to the value list *vs* for the current references

(st'.refs), and the current state of the FFI (st'.ffi). Function `do_app` fails if not enough or too many arguments to operator *op* are given in *vs*. Therefore the semantics raises a type error (`Rabort Rtype_error`) if `do_app` fails. If successful, function `do_app` returns a new state for the global references (*refs*), a new state of the FFI (*ffi*), and a value *v*. The overall result of the `evaluate` call is then the global state updated with *refs* and *ffi*, and value *v*.

Relaxed Floating-Point Semantics

Both the relaxed floating-point semantics and Icing use value trees to represent floating-point values. However, Icing's nondeterministic semantics cannot be directly added to CakeML source, because `evaluate` is a deterministic function. RealCake instead encodes the nondeterminism as a deterministic *optimization oracle*. Specifically, RealCake's relaxed floating-point semantics extends the global state with a floating-point optimization oracle:

```
fpState = <| rewrites : optimization list; opts : num → rewriteApp list;
             canOpt : optChoice; choices : num|>
```

The oracle stores the currently allowed optimizations in the field `rewrites`. Component `opts` encodes the oracle decisions of when which optimization is applied. `opts 0` returns all optimizations that are applied next during evaluation of a floating-point expression. The optimization scope `canOpt` models the fine-grained control by recording the last optimization scope that has been seen while evaluating. The relaxed floating-point semantics optimizes only if `canOpt` is an `opt`:annotation. In `choices` we track the number of optimizations that have been applied. We will use this global counter for integrating the relaxed floating-point semantics with the proof-producing synthesis.

In principle, RealCake's relaxed floating-point semantics and the Icing semantics model the same set of optimization results, as for each nondeterministic Icing result there exists a deterministic oracle under which RealCake's semantics returns the same value, and vice versa. However, supporting floating-point optimizations in CakeML source is only possible with the optimization oracle. Adding the oracle to the global state of the semantics causes the least amount of friction with existing CakeML proofs, while also enabling the nondeterministic simulation proofs from Icing in CakeML via manipulation of the global optimization oracle.

To integrate the relaxed floating-point semantics of RealCake with `evaluate`, Figure 5c adds a separate case to `evaluate` for floating-point operators. As for standard operator evaluation in Figure 5a, when evaluating a floating-point operation, `evaluate` first evaluates the arguments, and runs `do_app`. When evaluating a floating-point operation, function `do_app` does not alter the global state (st'.refs), and it does not call into the foreign function interface (st'.ffi). It simply returns the value tree representing operator *op* applied to argument values in *vs*. If `do_app` successfully returns value tree *r*, `evaluate` attempts to optimize the value tree. To this end, function `optimizeIfOk` first checks whether the `canOpt` field of the optimization oracle is set to `opt`. If optimizations are allowed, the function performs the optimizations of the oracle in field `opts`. Then, the optimization oracle is advanced to the next decision, and the global optimization counter `choices` is incremented. Function `optimizeIfOk` returns both the global state updated with the new optimization oracle, and the optimized value tree. If no optimizations are allowed, the function leaves its inputs unchanged. Finally, if `op` is a Boolean comparison of floating-point value trees (`isFpBool op`), `evaluate` turns the resulting value tree into a Boolean constant as CakeML eagerly evaluates control-flow expressions.

For the loose connection between Icing and CakeML, it was sufficient for Icing to turn value trees into floating-point words once a control-flow decision was made. To keep the changes to the CakeML semantics local and manageable, RealCake's relaxed floating-point semantics eagerly evaluates value trees into words as soon as a Boolean comparison is applied to them, even if no control-flow decision is made afterwards.

Figure 5b adds the optimization annotations `opt:` and `noopt:` as a separate case to `evaluate`. `FpOptimize annot e` means that expression e is evaluated under the optimization scope *annot*, which can either be `opt:` or `noopt:`. Evaluation of an optimization scope replaces the current semantic scope in `canOpt` with the new scope annotation (`updateOptFlag st annot`), before evaluating e. Next, the old annotation is recovered by `resetOptFlag` st' st. Function `addAnnot` *annot vs* ensures that all value trees in `vs` are extended with a correct scoping annotation. This is required to ensure that the semantics respects the fine-grained control. If `evaluate` did not add the annotation to the value trees, the semantics could optimize expression `noopt:(x + 2.4)` by first evaluating `x + 2.4` and then optimizing it once the expression is used as part of a larger floating-point expression.

4.3 Integrating Relaxed Floating-Point Semantics into the Compiler Toolchain

If we want to fully integrate RealCake's relaxed floating-point semantics with CakeML, we have to also integrate it with the tools included in the CakeML compiler toolchain. In the toolchain, a binary implementation of the compiler is obtained by verified bootstrapping [29] of the in-logic compiler using proof-producing synthesis [2]. Furthermore, CakeML source code can be verified using CakeML's program verification tools that rely on characteristic formulae (CF) [22], allowing Hoare-logic like manual proofs (*e.g.*, for verification of non-terminating programs [46] or a proof checker for higher-order logic [1]). To prove whole-program specifications (Section 6), we integrate RealCake's relaxed floating-point semantics with the proof-producing synthesis and CF.

CakeML Compiler Backend

A key insight for getting the deterministic compiler proofs to interact nicely with the optimization oracles used in RealCake's relaxed floating-point semantics was to implement the fast-math optimizer as a *source-level* optimization pass, separate from the CakeML compiler backend. With our extension from Subsection 4.1, the CakeML compiler backend compiles deterministic 64-bit floating-point kernels to machine code and we reuse this infrastructure by adding a third optimization scope, `strict`, to the relaxed floating-point semantics. Intuitively, we use the `strict` annotation to completely disallow floating-point optimizations in the compiler backend, allowing us to preserve determinism of the source semantics for the correctness proofs.

Any program that is run with the `strict` annotation will never apply optimizations and perform only IEEE-754 correct arithmetic operations. The difference between a `strict` and a `noopt` annotation is that `strict` is "sticky" in the sense that if a program ever enters `strict` mode, evaluation becomes deterministic and cannot escape from it through successive `opt` annotations, while `noopt` and `opt` can be mixed freely; e.g. a program may be under a `noopt` scope, while parts of it are marked with `opt` to selectively apply optimizations.

Proof-Producing Synthesis and CF

The proof-producing synthesis and the CF are key components of the CakeML compiler, and required for bootstrapping the compiler. As both crucially depend on how the CakeML source semantics are defined, we have to make sure that the bootstrapping still works, even after adding the relaxed floating-point semantics. Specifically, the synthesis relies on expressions being pure, and thus not altering global state. The crux is that we need the optimization oracle to reside in global state for the backwards simulation proofs, and therefore must ensure

that no floating-point optimizations can be applied in code produced by synthesis. The `choices` component of the optimization oracle makes optimization attempts by the semantics observable in the global state. We prove a lemma that if the optimization counter `choices` is not incremented, evaluation cannot have attempted to optimize floating-point code under an `opt`: scope. To use this lemma, the synthesis configures the initial optimization oracle to be running under an `opt` scope, with an empty list of optimization choices. From this configuration we show that no floating-point optimizations are ever attempted by synthesized code, reestablishing the invariant of the expression being pure.

We use an optimization counter instead of a Boolean flag, as some of our simulation theorems must combine optimization oracles, while preserving optimization decisions (*e.g.*, when combining oracles for left and right-hand sides of binary operators). In such proofs, the optimization counter gives an exact bound on when the behavior of the oracle must change.

The exact same technique is applied to CF: we make sure that programs reasoned about with CF cannot apply optimizations based on the optimization oracle.

4.4 Extending CakeML with Real-Number Arithmetic

The third semantics added to CakeML in RealCake is a real-number semantics used for bounding roundoff errors of floating-point kernels. We extend the CakeML source semantics with support for real numbers and real-number operations by adding a new case to `evaluate`'s operator evaluation in Figure 5d. Here, we focus on the real-number semantics. In Section 6 we explain how RealCake translates floating-point programs into their real-number counterpart.

Evaluation of real-number operations follows the simple case from Subsection 4.2. The main difference is that we extend the optimization oracle in the global state with an additional flag `real_sem`. Function `realsAllowed st.fp_state` checks that the flag is set to true, otherwise evaluation is aborted. The flag disallows real-number operations where necessary, as the real-valued semantics is only used for verification purposes. Further, the compiler does not compile real-valued operations or constants. In the compiler proofs, we rule out real-number operations by assuming that the flag is switched off.

We have presented operator evaluation separately. In our implementation, when evaluating an `App` *op es* expression, the CakeML source semantics first does a case split on *op* and chooses whether to apply standard operator evaluation (Figure 5a), relaxed floating-point semantics (Figure 5c), or real-number semantics (Figure 5d).

When integrating the relaxed floating-point semantics with proof-producing synthesis of CakeML (Subsection 4.3), the global counter `choices` is used to make attempted floating-point optimizations observable. To preserve invariants of the proof-producing synthesis, the real-number semantics requires a similar treatment: The global counter `choices` is incremented if evaluation of a real-number operation is attempted but fails (`advanceOracle`).

5 RealCake's Floating-Point Optimizer

In this section, we implement a fast-math-style peephole optimizer for RealCake and prove it correct with respect to the relaxed floating-point semantics. At a high-level, we split optimization into two steps. In step one, function `planOpts` computes which optimizations should be applied to the kernel. We call the list of optimizations returned by `planOpts` the *optimization plan* and refer to this first step as *optimization planning*. In step two, function `applyOpts(plan,e)` applies the optimization plan `plan` to floating-point kernel `e`. Before returning, the `noOpts` function tags the result with a marker to dissallow further optimizations, which is required to recover the determinism needed by the CakeML compiler proofs. We call this second step *optimization execution*.

$$x \times 0 \to 0$$
$$x \times 1 \to x$$
$$x \times -1 \to -x$$

$$x \times 2 \to x + x^*$$
$$x \times 3 \to x + (x + x)$$
$$x + 0 \to x$$
$$x - x \to 0$$

$$-(x \times y) \to x \times (-y)^*$$
$$x + (-y) \to x - y^*$$
$$x \times y + z \to \texttt{fma}(x, y, z)$$

Figure 6 Optimizations currently used by the peephole optimization phase, IEEE-754 preserving optimizations are marked with a $*$.

Optimization Planning

For a floating-point kernel `e`, function `planOpts(e)` returns a list of tuples `(path, opts)`, where the left-hand side `path` is an index into the kernel stating *where* the kernel should be optimized, and the right-hand side `opts` is a list of optimizations stating *how* the kernel should be optimized. The optimization planner `planOpts` is split into the following phases (applied in this order):

- `canonicalForm` puts all floating-point kernels into a canonical shape replacing $x - y$ with $x + ((-1) \times y)$, associating $+, \times$ to the right ($(x + y) + z \to x + (y + z)$), and moving constants to right-hand sides with commutativity of $+$ and $\times$.
- `undistribute` replaces expressions like $(x \times y) + (x \times z)$ with $x \times (y + z)$, "undistributing" as much as possible to increase possibilities for FMA-introduction, and reduce the size of the floating-point kernel. The symmetric case of $(y \times x) + (z \times x)$ is ignored by the undistribute phase, as `canonicalForm` rotates all multiplications with commutativity.
- `peepholeOptimize` re-establishes canonical form and applies the optimizations from Figure 6.
- `balanceTrees` reorders sub-expressions in the floating-point kernel by replacing deeply-nested arithmetic expressions like $x_1 + (x_2 + (x_3 + x_4))$ by more shallow versions, such as $(x_1 + x_2) + (x_3 + x_4)$ and similarly for $\times$.[8]

Function `composePlans` concatenates the optimization plans produced by each phase.

Optimization Execution

When executing the optimization plan, function `applyOpts` first runs function `optimizeWithPlan` on the plan and its input kernel, where `optimizeWithPlan` applies all elements of a given optimization plan one by one. Function `optimizeWithPlan` optimizes an expression only if it is wrapped under an `opt:` annotation. Further, either all or none of the optimizations in the plan are applied: if optimization fails, then the unoptimized input kernel is returned.

For each element of the plan `(path, opts)`, `optimizeWithPlan` traverses expression `e` following `path` until reaching a sub-expression `e'` and applies the optimizations `opts` at the end of the path. Having reached expression `e'` at the end of `path`, function `optimizeWithPlan` calls function `rewrite(e, opts)` that applies the optimizations `opts` to the CakeML expression `e'`.

As CakeML source supports stateful features like reference cells, and calls into a foreign-function-interface (FFI), function `rewrite(e, opts)` checks that CakeML expression `e` is a pure (floating-point) expression. This check, which is implemented as a function `isPureExp(e)`, effectively rules out optimization of expressions that use any of CakeML's stateful features.

[8] We added `balanceTrees` as an optimization pass to simplify register allocations.

The result of running `optimizeWithPlan` is given to function `noOpts`, that performs a bottom-up traversal of expression `e`, replacing any `opt:` annotation with a `noopt:` annotation, disallowing further optimizations and, as a result, making the program's semantics deterministic.

5.1 Correctness of the Fast-Math Optimizer

Our optimizer is split into two separate phases, optimization planning, and optimization execution. A key benefit of this split is that we can prove correctness of optimization execution without caring about the exact optimizations contained in the plan. Rather, we verify `applyOpts` for any potential plan generated by our optimization planner.

At a high-level, we show that the optimizations done by `applyOpts` are correct with respect to the relaxed floating-point semantics, and no further optimizations can be applied afterwards. Accordingly, we split correctness of `applyOpts` into two proofs. First, we prove that running the result of `noOpts`(e) gives the same result as running `e` with an oracle that performs no optimizations. The correctness proof for `noOpts`(e) is a simple backwards simulation and thus we do not show it here. Second, we prove that there is a backwards simulation between the result of `optimizeWithPlan(e, plan)` and `e`, where `plan` has been generated by our planner.

▶ **Theorem 2** (`optimizeWithPlan` – correctness)**.**

$$\begin{array}{l}
\texttt{evaluate}\ st_1\ env\ (\texttt{optimizeWithPlan}\ (\texttt{planOpts}\ e)\ exps)\ =\ (st_2, \texttt{Rval}\ r)\ \wedge \\
\texttt{allVarsBoundToFPVal}\ \ exps\ env \wedge\ \texttt{flagAndScopeAgree}\ cfg\ st_1.\mathsf{fp_state}\ \wedge \\
\texttt{notInStrictMode}\ st_1.\mathsf{fp_state}\ \wedge\ \texttt{noRealsAllowed}\ st_1.\mathsf{fp_state}\ \Rightarrow \\
\exists fpOpt\ choices\ fpOptR\ choicesR. \\
\quad \texttt{evaluate}\ (\texttt{appendOptsAndOracle}\ st_1\ (\texttt{getRws}\ (\texttt{planOpts}\ e))\ fpOpt\ choices)\ env\ exps\ = \\
\quad\ (\texttt{appendOptsAndOracle}\ st_2\ (\texttt{getRws}\ (\texttt{planOpts}\ e))\ fpOptR\ choicesR, \texttt{Rval}\ r)
\end{array}$$

Theorem 2 proves: for the result obtained from evaluating the syntactically optimized kernel, there exists an optimization oracle such that `evaluate` returns the same result when semantically optimizing with the optimizations from the computed plan. The CakeML source semantics is untyped, and thus we assume that all variables are bound to floating-point constants in `exps` (`allVarsBoundToFPVal`). Instead of showing correctness of `optimizeWithPlan` for the overall plan, we reduce the global correctness proof to a series of correctness proofs about the separate phases, and combine them into the overall backwards simulation.

Extending the Optimizer

Extending the RealCake optimizer requires extending both the implementation of the optimizer, and its correctness proof. To add a new peephole optimization, a user adds the optimization to the list of optimization of `peepholeOptimize`, and extends the correctness theorem for `peepholeOptimize`. All other theorems need not be changed. We provide a set of lemmas that can be used to reduce the global correctness proof of `peepholeOptimize` to a simple local backwards simulation for the newly-added optimization in terms of the `rewrite` function only. Adding a new phase to `planOpts` is more involved as it requires showing a global correctness theorem for the newly added phase, as well as extending the theorem that splits up correctness of `planOpts` into correctness of its components. The complexity of the first proof depends on the complexity of the phase, whereas splitting up the correctness proof for `planOpts` is a straightforward proof showing that optimizations of the newly added phase are contained in the optimizations applied by `planOpts`.

6 Proving Error Refinement with RealCake

CakeML with relaxed floating-point semantics optimizes floating-point kernels and automatically proves a relation between the unoptimized and the optimized kernel. However, to meaningfully support floating-point arithmetic in a verified compiler, the compiler must relate the unoptimized real-valued program and the optimized floating-point program.

Classic compiler optimizations like constant propagation and dead-code elimination have a clear definition of when they can be applied and one can prove that the optimizations do not change the program result. Floating-point fast-math optimizations do not follow this intuition in general. As an example, we can introduce an FMA instruction in the simple expression `x * 2.9 + 0.05` with relaxed floating-point semantics: `fma(x, 2.9, 0.05)`. The FMA makes the expression generally faster and locally more accurate, as the result is only rounded once. Correctness of the fast-math optimizer proves a backwards simulation between the expressions, however, the theorem does not capture the change in roundoff errors.

We propose the notion of *error refinement*: the compiler may optimize a floating-point kernel aggressively as long as the results remain within a (given) bound relative to *real-number* semantics. This notion captures the implicit assumption or expectation by the programmer. We make this notion of error refinement explicit by implementing a fully automatic pipeline that computes an upper bound on the roundoff error of a floating-point kernel in CakeML and compares it to a user-specified accuracy bound. For this we use the roundoff error analysis tool FloVer [6], implemented in HOL4. We prove the roundoff error bound correct with respect to a run of the original input kernel under an idealized real-number semantics.

6.1 Translating RealCake Kernels into FloVer Input

To infer roundoff errors for a RealCake kernel with FloVer, we define a straightforward encoding function `toFloVer`(e), translating floating-point kernels with variables, constants, unary and binary floating-point operations, FMAs, and let bindings into FloVer syntax. Correctness of the translation functions proves once and for all a simulation relating deterministic RealCake floating-point semantics with FloVer's idealized finite-precision semantics. To prove the simulation, our translation function ensures that the kernel is wrapped under a `noopt:` annotation. As roundoff error analysis tools depend on ranges for the input variables our pipeline also requires a real-number function specifying these input constraints.

RealCake implements a function `isOkError(e, P, err)` that returns true if `err` is a sound upper bound on the worst-case roundoff error for RealCake expression `e` and input constraints `P`. First, the RealCake kernel `e` is translated into FloVer syntax with `toFloVer`(e). Function `isOkError` then runs FloVer's unverified inference algorithm to generate a (untrusted) roundoff error analysis certificate for the FloVer encoding of `e` and input constraints `P`. FloVer's certificate checker automatically checks the certificate, and if the check suceeds, the error bound encoded in the certificate is correct. Finally, `isOkError` checks that the global upper bound encoded in the certificate is smaller or equal to the user-specified error constraint `err`.

6.2 Proving Roundoff Error Bounds for RealCake Kernels

To prove error refinement for an optimized kernel, we connect the soundness theorem of FloVer to RealCake's relaxed floating-point semantics. Together with the idealized real-valued semantics we show once and for all the HOL4 theorem:

▶ **Theorem 3** (CakeML-FloVer roundoff errors)**.**

$\forall f\ P\ err\ theVars\ vs\ body\ env.$
`isOkError_succeeds` $(f, P, err, theVars, body)\ \wedge$ `isPrecondFine` $theVars\ vs\ P\ \Rightarrow$
$\exists\, r\ fp.$
`realEvals_to` (`realify` $body$) (`envWithRealVars` $env\ theVars\ vs$) $r\ \wedge$
`floatEvals_to` $body$ (`envWithFloatVars` $env\ theVars\ vs$) $fp\ \wedge$
`abs` (`valueTree2real` $fp\ -\ r$) $\leq\ err$

On a high-level, Theorem 3 states that if function `isOkError` succeeds, the analyzed function can be run both under floating-point and real-number semantics, and `err` is an upper bound on the roundoff error. The assumptions are: `isOkError` succeeds, and `body` is the function body of RealCake floating-point kernel `f`, with the parameters `theVars` (`isOkError_succeeds`); and the values vs bound to the parameters $theVars$ are within the input constraints P (`isPrecondFine` $theVars\ vs\ P$). `realify` replaces floating-point operations by their real-number counterparts. The theorem shows that there exists a real number r and a floating-point word fp such that evaluation of the function under an idealized real-number semantics returns r (`realEvals_to`), evaluation under floating-point semantics returns fp (`floatEvals_to`), and err is an upper bound to the roundoff error of function f (`abs`(`valueTree2real` $fp - r) \leq err$).

Error refinement relates the user-given error bound back to a real-number semantics of the initial, unoptimized kernel, but RealCake runs function `isOkError` on the optimized kernel. In addition to Theorem 3 we also need to prove that the applied optimizations are real-valued identities. Exactly like we prove correctness of `optimizeWithPlan` in Subsection 5.1, we have proven once and for all a simulation between the real-number semantics of the optimized kernel and its unoptimized version. Combining this theorem with Theorem 3, we automatically prove error refinement for floating-point kernels.

7 Evaluation: Performance and Accuracy Proofs

The RealCake development spans roughly 35k lines of proof-code, composed of the IEEE floating-point implementation and proofs, including the ARMv7 backend (∼1.5k LOC), the relaxed floating-point semantics and the real-number semantics (∼7k LOC, including proofs), the implementation and correctness proofs for the optimizer (∼20k LOC), and the benchmarks from the evaluation (∼7k LOC).

We evaluate RealCake on 51 benchmarks taken from the standard floating-point benchmark set FPBench [14]. Our evaluation includes all FPBench benchmarks that use floating-point operations that are supported by RealCake and we exclude only those that cannot be expressed in RealCake (for instance we exclude benchmarks with elementary function calls; *i.e.* functions like sin, cos, . . .). We use the preconditions that are already specified in FPBench, but modify them slightly for the `jetEngine` and `n_body` kernels such that FloVer can prove a roundoff error bound and does not report a possible division by zero. Our evaluation shows how RealCake establishes end-to-end correctness proofs, and compares the runtime of the optimized and unoptimized kernels.

Table 1 Roundoff errors for optimized and unoptimized FPBench benchmarks; benchmarks where the roundoff error improves are highlighted in bold font and benchmarks where no end-to-end specification is proven are underlined.

Name	Orig	fast-math	Impr.
bspline3	1.295e-16	1.295e-16	0%
carbonGas	5.688e-08	5.688e-08	0%
cartToPol	2.815e-09	2.463e-09	13%
delta4	4.048e-12	2.028e-13	75%
delta	1.970e-13	2.940e-12	-198%
doppler1	6.534e-13	6.412e-13	2%
doppler2	6.534e-13	1.639e-12	50%
doppler3	1.675e-12	2.680e-13	20%
himmilbeau	3.417e-12	3.003e-12	12%
hypot	2.815e-09	2.463e-09	13%
hypot32	2.815e-09	2.463e-09	13%
i4modified	4.002e-13	4.002e-13	0%
intro_ex	2.220e-10	2.220e-10	0%
jetEngineModi	5.209e-08	3.898e-08	25%
kepler0	1.761e-13	1.801e-13	-2%
kepler1	8.397e-13	8.467e-13	-1%
kepler2	4.069e-12	3.973e-12	2%
matDet2	5.107e-12	4.663e-12	9%
matDet	5.107e-12	4.663e-12	9%
n_bodyXmod	ERR	ERR	ERR
n_bodyZmod	ERR	ERR	ERR
nonlin1	2.220e-10	2.220e-10	0%
nonlin2	2.657e-09	2.657e-09	0%
pid	7.621e-15	7.727e-15	-1%
predatorPrey	3.395e-16	3.366e-16	1%
rigidBody1	6.565e-11	5.329e-13	80%

Name	Orig	fast-math	Impr.
rigidBody2	5.579e-13	6.410e-11	-360%
rump_C	4.079e+22	3.859e+22	5%
rump_rev	3.859e+22	3.679e+22	5%
rump_pow	4.079e+22	3.859e+22	5%
runge_kutta_4	2.220e-14	2.220e-14	0%
sec4_example	2.657e-09	2.657e-09	0%
sine_newton	7.495e-15	6.275e-15	16%
sineOrder3	1.765e-15	1.765e-15	0%
sine	1.538e-15	1.373e-15	11%
sqroot	1.115e-15	1.059e-15	5%
sqrt_add	1.322e-12	1.322e-12	0%
sum	5.995e-15	5.995e-15	0%
t01_s3	5.995e-15	5.995e-15	0%
t02_s8	9.548e-15	8.438e-15	12%
t03_nl2	4.885e-14	4.885e-14	0%
t04_dqmom9	1.999	1.999	0%
t05_nl1_r4	4.441e-06	4.441e-06	0%
t05_nl1_t2	2.776e-16	2.776e-16	0%
t06_sums4_sum1	1.443e-15	1.332e-15	8%
t06_sums4_sum2	1.332e-15	1.332e-15	0%
turbine1	1.588e-13	1.541e-13	3%
turbine2	2.213e-13	2.213e-13	0%
turbine3	1.108e-13	1.061e-13	4%
verhulst	8.343e-16	8.343e-16	0%
x_by_xy	2.220e-15	2.220e-15	0%

7.1 Automated End-To-End Proofs

We have translated all 51 FPBench benchmarks into HOL4 script files that are read by RealCake. Each script file defines the original, unoptimized, floating-point kernel, a precondition for the kernel, and a user-provided error bound. For simplicity, our evaluation uses 2^{-5} as the user-provided error bound for all of the benchmarks, though those would be given by the compiler user in a real-world setting.[9]

Our HOL4 automation at the end of each script file fully automatically optimizes the kernel, instantiates Theorem 2 for the generated plan, infers a roundoff error bound and compares it to the user-provided error bound. Finally, a whole-program specification relating the behavior of the machine code for the optimized program to the real-number semantics of the unoptimized program is proven automatically by combining the individual proofs.

RealCake proves the end-to-end correctness theorem (Theorem 1) for 45 benchmarks. That is, for these benchmarks it is able to show that the roundoff error of the optimized program is below the specified default error bound of 2^{-5}. For the three `rump` benchmarks and the `test04_dqmom9` benchmark, the computed errors are larger than the user-provided error bound (already for the original unoptimized program), and for the benchmarks `n_bodyXmod` and `n_bodyZmod` FloVer is not able to infer a roundoff error bound as its HOL4 computation becomes stuck, likely due to limitations in the HOL4 real number computations.

[9] If the error bound is choosen too tightly the optimizer may reject every optimization candidate, while a too coarse bound could allow for too aggressive optimizations.

Table 2 Running times for optimized and unoptimized FPBench benchmarks on the Raspberry Pi v3; benchmarks where performance improves *with fast-math optimizations* are highlighted in bold.

Name	Orig	Csts	Csts + fast-math	Name	Orig	Csts	Csts + fast-math
bspline3*	18.14	1.75 (91%)	1.75 (91% / 0%)	**rigidBody2**	54.92	5.10 (91%)	4.54 (92% / 11%)
carbonGas *	103.40	3.85 (97%)	3.85 (97% / 0%)	**rump_C**	107.48	6.82 (94%)	6.26 (95% / 9%)
cartToPol	2.05	2.04 (1%)	1.86 (10% / 9%)	**rump_rev**	107.96	6.80 (94%)	6.27 (95% / 8%)
delta4	6.34	6.33 (1%)	6.17 (3% / 3%)	**rump_pow**	112.54	12.34 (90%)	11.57 (90% / 7%)
delta	13.49	13.47 (1%)	11.44 (16% / 16%)	runge_kutta_4*	93.46	9.53 (90%)	9.53 (90% / 0%)
doppler1	36.02	3.25 (91%)	3.06 (92% / 6%)	sec4_example*	34.99	2.40 (94%)	2.40 (94% / 0%)
doppler2	36.00	3.25 (91%)	3.06 (92% / 6%)	sineOrder3*	34.86	2.08 (95%)	2.08 (95% / 0%)
doppler3	35.98	3.25 (91%)	3.07 (92% / 6%)	sine_newton*	126.34	10.73 (92%)	10.73 (92% / 0%)
himmilbeau	36.13	3.36 (91%)	3.05 (92% / 10%)	sine*	55.36	6.03 (90%)	6.03 (90% / 0%)
hypot32	2.04	2.04 (1%)	1.86 (10% / 9%)	**sqroot**	87.06	4.85 (95%)	4.65 (95% / 5%)
hypot	2.05	2.05 (1%)	1.86 (10% / 10%)	sqrt_add*	35.21	2.59 (93%)	2.59 (93% / 0%)
i4modified*	1.77	1.78 (0%)	1.78 (0% / 0%)	sum*	3.07	3.07 (1%)	3.07 (1% / 0%)
intro_ex*	17.73	1.32 (93%)	1.32 (93% / 0%)	t01_s3*	3.07	3.08 (0%)	3.08 (0% / 0%)
jetEngineMod	195.99	11.89 (94%)	11.12 (95% / 7%)	t02_s8*	3.04	3.05 (0%)	3.05 (0% / 0%)
kepler0	5.32	5.31 (1%)	5.30 (1% / 1%)	t03_nl2*	1.78	1.78 (1%)	1.78 (1% / 0%)
kepler1	8.19	8.20 (0%)	8.16 (1% / 1%)	**t04_dqmom9**	163.82	11.76 (93%)	10.20 (94% / 14%)
kepler2	12.43	12.41 (1%)	12.22 (2% / 2%)	t05_nl1_r4*	34.67	2.06 (95%)	2.06 (95% / 0%)
matDet2	6.38	6.37 (1%)	5.67 (12% / 12%)	t05_nl1_test2*	34.00	1.54 (96%)	1.54 (96% / 0%)
matDet	6.37	6.38 (0%)	5.65 (12% / 12%)	t06_sums4_sum1*	1.70	1.70 (0%)	1.70 (0% / 0%)
n_bodyXmod	38.46	5.20 (87%)	5.06 (87% / 3%)	t06_sums4_sum2*	1.68	1.67 (1%)	1.67 (1% / 0%)
n_bodyZmod	38.40	5.27 (87%)	5.15 (87% / 0%)	turbine1*	121.02	5.29 (96%)	5.29 (96% / 0%)
nonlin1*	17.72	1.31 (93%)	1.31 (93% / 0%)	turbine2*	69.90	3.94 (95%)	3.94 (95% / 0%)
nonlin2*	35.06	2.41 (94%)	2.41 (94% / 0%)	turbine3*	121.29	5.28 (96%)	5.28 (96% / 0%)
pid*	104.11	4.72 (96%)	4.72 (96% / 0%)	verhulst*	51.28	2.27 (96%)	2.27 (96% / 0%)
predatorPrey	52.25	2.81 (95%)	3.08 (95% / -9%)	x_by_xy*	1.51	1.51 (1%)	1.51 (1% / 0%)
rigidBody1*	19.11	2.78 (86%)	2.78 (86% / 0%)				

We show the errors for the optimized and unoptimized kernels in Table 1. “Orig.” is the roundoff error for the unoptimized kernel, and “fast-math” is the roundoff error for the optimized kernel, and column “Impr.” shows the percentage by which the error improved with our fast-math optimizations, *i.e.* if the number is less than 0% the error has increased, and decreased if it is greater than 0%. We highlight benchmarks where the roundoff error has been decreased by the RealCake optimizer in bold font. While improving the roundoff error is not the goal of our optimizations, FMA instructions are said to be locally more accurate, and reordering of operations influences roundoff errors too. Hence we evaluate the effect on roundoff errors of our optimization strategy. The benchmarks `delta4`, `delta`, `rigidBody1`, and `rigidBody2` have the largest difference in roundoff errors. By inspecting the generated code we found that in these cases, RealCake has significantly alterted the structure of the kernel. The roundoff error computed for a single kernel is highly influenced by the order of operations, thus we suspect that this large difference is mainly due to operator odering. Overall, we notice that if RealCake can infer a roundoff error, the error of the optimized kernel is usually within the same order of magnitude as the unoptimized version, but in many cases it is actually more accurate.

7.2 Performance Improvements

We compared the performance of unoptimized and RealCake’s optimized floating-point kernels. In a first run, we measured wide differences in speedups and slowdowns. By manually inspecting the code, we noticed a missing optimization in CakeML: 64-bit word constants should be pre-allocated (or lifted) to increase performance. Lifting constants is a worthwhile optimization in general, and particularly effective for floating-point programs, as

it is does not change the program's IEEE-754-semantics and floating-point programs usually contain many constants. Thus, we implemented an independent, semantics preserving, global optimization that preallocates 64-bit words as global variables. Our performance evaluation compares three versions of FPBench kernels: the unoptimized version as a baseline, the kernel with preallocated constants, and the kernel after first applying fast-math optimizations and then preallocating constants.

To measure performance, CakeML generates ARMv7 machine code where each numerical kernel is run 10 million times in a loop. Each version of the benchmark, with the core loop running the kernel 10 million times, is run three times on five different sets of inputs, for a total of fifteen runs per benchmark.

We run the ARMv7 code on a Raspberry Pi v3 and summarize the results in Table 2. Column "Orig." shows the running time of the (10 million iterations of the) unoptimized program in seconds. Column "Csts." shows the running time of the program with preallocated constants in seconds plus the relative speedup in percent. And column "Csts. + fast-math" shows the running time of the program when first running RealCake's optimizer and then preallocating constants in seconds plus first the relative speedup in percent with respect to the unoptimized program, and second the relative speedup with respect to the version with preallocated constants. We mark benchmarks with a performance improvement of more than 1% of the fast-math optimizations with respect to preallocating constants in bold (we identified a difference within ±1% to be noise).

Initially, some benchmarks experienced slowdowns of up to 20%. Via manual inspection, we noticed that the fast-math optimizer created too many instructions. As a simple heuristic to prevent this problem, RealCake sums the arities of the floating-point operators in the program versions, and returns the unoptimized version if the heuristic value of the fast-math optimized program is greater than or equal to the unoptimized program. Even if the heuristic rejects an optimization, RealCake computes roundoff errors for both program versions and proves an end-to-end specification theorem about the optimized program. In total, the heuristic rejects optimizations for 27 benchmarks, and we mark them with a $*$ in Table 2.

Overall the evaluation shows that preallocating constants is a valuable optimization for CakeML on its own. On top of this, our fast-math optimizer is able to improve the performance for 20 benchmarks and for 7 of those significantly ($> 10\%$). This is remarkable, since the FPBench benchmarks are carefully hand-written and do not target optimizations specifically and are not representative of, e.g., automatically generated code from tools such as Matlab that would be used in the development of embedded system kernels.

For one benchmark we notice a slowdown of 9% even with our heuristic, and the program versions differ only by a single FMA instruction. We suspect that this slowdown is due to bad pipelining on the Raspberry Pi.

RealCake's constant preallocation achieves a geometric mean speeup of 83%, and the geometric mean of the speedup for RealCake's optimizer compared with the program with preallocated constants is 3%. The maximum speedup achieved with preallocating constants only is 97%, and we notice no slowdowns. When applying fast-math optimization, the greatest slowdown is -9%, and the maximum speedup is 16%.

In general, benchmarks with higher speed-ups from our optimization strategy usually provide many opportunities to both introduce `fma` instructions, and remove constants. We think that the foundational work in RealCake facilitates exploration of other optimization strategies in the future.

8 Related Work

Verified Compilation of Floating-Point Programs

Besides CakeML, CompCert [31] is the other major available verified compiler, compiling imperative C programs. CompCert supports floating-point programs [10] following the strict IEEE-754 semantics. This semantics allows it to perform a few small optimizations that are IEEE-754 compliant such as constant propagation and replacing a multiplication by two by an addition ($x \times 2 \rightarrow x + x$).

RealCake supports additional optimizations based on real-valued identities that are not IEEE-754 compliant. While our implementation is done in the context of CakeML and verified in HOL4, the principles of RealCake are independent of the particular programming language that is being compiled and should thus be portable to CompCert as well.

The Alive framework [34] provides a way to specify and prove correct peephole optimizations for C++ code that can be applied in an LLVM pass. Alive verifies optimizations using SMT solvers and has been extended to bit-precise floating-point optimizations and optimizations involving special values, satisfying the IEEE-754 standard [39, 42]. These optimizations are complementary to RealCake's optimizations. Formal verification of Alive's peephole optimizations is addressed separately by the AliveInLean project [30]. The VELLVM project [59] provides a rigorous semantics for LLVM IR semantics to reason about optimizations and implements IEEE-754-preserving floating-point arithmetic.

Verification of Floating-point Programs

Besides FloVer, there are several other tools that provide formally verified roundoff error bounds for floating-point arithmetic expressions:
FPTaylor [54], real2Float [36], Precisa [41], Gappa [17], and each of these can in principle replace FloVer in RealCake. We chose FloVer for convenience as it is implemented in HOL4.

Verification of floating-point programs that go beyond numerical kernels is still relatively limited. The above-mentioned automated tools, for instance, do not consider function calls, and techniques for loops are very restricted [41, 15], and thus require the user to provide range annotations for each function call, as well as loop invariants in general. Entire programs have been manually formally verified w.r.t. a real-valued specification, but with considerable human effort [48, 9], which is not suitable for a compilation setting.

If we only require verification of runtime exceptions, resp. absence of special values, then abstract interpretation-based techniques do scale to larger programs [27] and some provide formal verification [28].

Optimization of Floating-Point Programs

Floating-point optimizations have also been considered outside of the traditional compiler context, most of them focused on performance optimization.

Precimonious [50] performs mixed-precision tuning, by determining which operations can be implemented in a lower or higher precision, while satifying a user-provided error bound. While Precimonious can handle short programs with loops, it cannot guarantee the error bound as it uses a dynamic error analysis. Both FPTuner [11] and Daisy [16] perform mixed-precision tuning while providing accuracy guarantees using a static analysis, but can only handle relatively short straight-line expressions. Mixed-precision tuning requires a global error analysis and is thus not suitable to be performed inside a fundamentally modular

compiler. However, the precision-tuned program could be further optimized by a (verified) compiler. While RealCake currently only supports double precision floating-point arithmetic, an extension with (uniform) single precision requires merely some engineering work.

Several tools improve the performance or accuracy of floating-point programs by rewriting with real-valued identities. Spiral [47] rewrites linear algebra kernels to improve their performance on a particular hardware platform. Spiral does not take into account roundoff errors; its rewrites are not IEEE-754-preserving, but it does not quantify the errors. The HELIX project [58] uses Spiral as an external oracle for building a verified optimization pipeline for dataflow optimizations. Optimizations in HELIX are done with respect to real-number semantics which is orthogonal to RealCake's floating-point peephole optimizer.

Herbie [44] aims to improve the accuracy, instead of performance, of floating-point arithmetic expressions but estimates roundoff errors unsoundly using a dynamic analysis. The Salsa [13] tool applies a set of transformation rules to improve performance while soundly tracking roundoff errors. Finally, Daisy [16] first applies rewriting similar to Herbie in order to improve performance gains due to mixed-precision tuning. Still further away is the tool STOKE [52], which generates small floating-point kernels by superoptimization, but which does not even guarantee real-valued equivalence. We consider these optimizations to be orthogonal to the fast-math optimizations that we consider in RealCake. We note that the scoping mechanism allows RealCake to easily integrate parts of the code that have been heavily optimized, and that thus should not be modified further by the compiler.

9 Conclusion

We have presented RealCake, an extension of the CakeML compiler with fast-math-style floating-point optimizations. Using an oracle-based relaxed floating-point semantics we have integrated nondeterminstic semantics for fast-math-style optimizations into the verified CakeML compiler. Via a connection to an external accuracy analysis, RealCake establishes accuracy guarantees for the optimized program, relating it back to the real-numbered execution of the unoptimized program. In summary, RealCake is the first verified compiler that establishes end-to-end floating-point accuracy guarantees to enable fast-math-style optimization and prove end-to-end compilation theorems. Our evaluation has shown how RealCake automatically verifies whole programs, proving properties about their I/O behavior and accuracy. Further, both our fast-math optimizer and our global constant lifting achieve significant performance improvements. RealCake's error refinement establishes the core infrastructure necessary to verify fast-math-style peephole optimizations, enabling the implementation of additional optimizations such as vectorization in the future.

References

1. Oskar Abrahamsson. A Verified Proof Checker for Higher-Order Logic. *Journal of Logical and Algebraic Methods in Programming*, 112, 2020. `doi:10.1016/j.jlamp.2020.100530`.
2. Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *J. Autom. Reason.*, 64(7), 2020. `doi:10.1007/s10817-020-09559-8`.
3. A. Anta and P. Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *IEEE Transactions on Automatic Control*, 55(9):2030–2042, 2010. `doi:10.1109/TAC.2010.2042980`.
4. Apache Software Foundation. The LLVM Compiler Infrastructure, 2020. URL: `https://www.llvm.org/`.

5. Heiko Becker, Eva Darulova, Magnus O Myreen, and Zachary Tatlock. Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler. In *Computer Aided Verification (CAV)*, 2019. doi:10.1007/978-3-030-25543-5_10.
6. Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O Myreen, and Anthony Fox. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD)*, 2018. doi:10.23919/FMCAD.2018.8603019.
7. Michael Berg. LLVM Numerics Blog, 2019. URL: http://blog.llvm.org/2019/03/llvm-numerics-blog.html.
8. Hans-J. Boehm. Towards an API for the Real Numbers. In *Programming Language Design and Implementation (PLDI)*, 2020. doi:10.1145/3385412.3386037.
9. Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4), 2013. doi:10.1007/s10817-012-9255-4.
10. Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015. doi:10.1007/s10817-014-9317-x.
11. Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous Floating-Point Mixed-Precision Tuning. In *Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009846.
12. G. A. Constantinides, P. Y. K. Cheung, and W. Luk. Wordlength Optimization for Linear Digital Signal Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10), 2003. doi:10.1109/TCAD.2003.818119.
13. Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Improving the Numerical Accuracy of Programs by Automatic Transformation. *International Journal on Software Tools for Technology Transfer*, 19(4), 2017. doi:10.1007/s10009-016-0435-0.
14. Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification (NSV)*, 2016. doi:10.1007/978-3-319-54292-8_6.
15. Eva Darulova and Viktor Kuncak. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2), 2017. doi:10.1145/3014426.
16. Eva Darulova, Saksham Sharma, and Einar Horn. Sound Mixed-Precision Optimization with Rewriting. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2018. doi:10.1109/ICCPS.2018.00028.
17. Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted Verification of Elementary Functions Using Gappa. In *ACM Symposium on Applied Computing (SAC)*, 2006. doi:10.1145/1141277.1141584.
18. Anthony Fox. Improved Tool Support for Machine-Code Decompilation in HOL4. In *International Conference on Interactive Theorem Proving*. Springer, 2015. doi:10.1007/978-3-319-22102-1_12.
19. Free Software Foundation. The GNU Compiler Collection, 2020. URL: https://gcc.gnu.org/.
20. GCC Developers. GCC Wiki: Floating-point Math, 2020. URL: https://gcc.gnu.org/wiki/FloatingPointMath.
21. Alejandro Gomez-Londono, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do You Have Space for Dessert? A Verified Space Cost Semantics for CakeML Programs. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 4, 2020. doi:10.1145/3428272.
22. Armaël Guéneau, Magnus O Myreen, Ramana Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP)*, 2017. doi:10.1007/978-3-662-54434-1_22.

23 Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning (ICML)*, 2015.

24 John Harrison. Floating Point Verification in HOL. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1995. `doi:10.1007/3-540-60275-5_65`.

25 Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2nd edition, 2002. `doi:10.1137/1.9780898718027`.

26 IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019. `doi:10.1109/IEEESTD.2019.8766229`.

27 Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification (CAV)*, 2009. `doi:10.1007/978-3-642-02658-4_52`.

28 Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In *Principles of Programming Languages (POPL)*, 2015. `doi:10.1145/2676726.2676966`.

29 Ramana Kumar. *Self-Compilation and Self-Verification*. PhD thesis, University of Cambridge, 2015.

30 Juneyoung Lee, Chung-Kil Hur, and Nuno P Lopes. AliveInLean: a Verified LLVM Peephole Optimization Verifier. In *International Conference on Computer Aided Verification*. Springer, 2019. `doi:10.1007/978-3-030-25543-5_25`.

31 Xavier Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In *Principles of Programming Languages (POPL)*, 2006. `doi:10.1145/1111037.1111042`.

32 Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4), 2009. `doi:10.1007/s10817-009-9155-4`.

33 LLVM Developers. LLVM Language Reference: Fast-Math Flags, 2020. URL: `https://llvm.org/docs/LangRef.html#fast-math-flags`.

34 Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. In *Programming Language Design and Implementation (PLDI)*, 2015. `doi:10.1145/2737924.2737965`.

35 Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified Compilation on a Verified Processor. In *Programming Language Design and Implementation (PLDI)*, 2019. `doi:10.1145/3314221.3314622`.

36 Victor Magron, George Constantinides, and Alastair Donaldson. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Transactions on Mathematical Software*, 43(4), 2017. `doi:10.1145/3015465`.

37 Rupak Majumdar, Indranil Saha, and Majid Zamani. Synthesis of Minimal-Error Control Software. In *International Conference on Embedded Software (EMSOFT)*, 2012. `doi:10.1145/2380356.2380380`.

38 Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic Verification of Control System Implementations. In *International Conference on Embedded software (EMSOFT)*, 2010. `doi:10.1145/1879021.1879024`.

39 David Menendez, Santosh Nagarakatte, and Aarti Gupta. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis Symposium (SAS)*, 2016. `doi:10.1007/978-3-662-53413-7_16`.

40 Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009. `doi:10.1137/1.9780898717716`.

41 Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Computer Safety, Reliability, and Security (SAFECOMP)*, 2017. `doi:10.1007/978-3-319-66266-4_14`.

42 Andres Nötzli and Fraser Brown. LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM. In *International Workshop on State Of the Art in Program Analysis (SOAP)*, 2016. doi:10.1145/2931021.2931024.

43 Scott Owens, Magnus O Myreen, Ramana Kumar, and Yong Kiam Tan. Functional Big-Step Semantics. In *European Symposium on Programming (ESOP)*, 2016. doi:10.1007/978-3-662-49498-1_23.

44 Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically Improving Accuracy for Floating Point Expressions. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015. doi:10.1145/2737924.2737959.

45 A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1998. doi:10.1007/BFb0054170.

46 Johannes Aman Pohjola, Henrik Rostedt, and Magnus O. Myreen. Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs. In *Interactive Theorem Proving (ITP)*, 2019. doi:10.4230/LIPIcs.ITP.2019.32.

47 Markus Püschel, José M F Moura, Bryan Singer, Jianxin Xiong, Jeremy R Johnson, David A Padua, Manuela M Veloso, and Robert W Johnson. Spiral - A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *International Journal of High Performance Computing Applications*, 18(1), 2004.

48 Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *Certified Programs and Proofs (CPP)*, 2016. doi:10.1145/2854065.2854066.

49 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends in Programming Languages*, 5, 2019. doi:10.1561/2500000045.

50 Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning Assistant for Floating-Point Precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013. doi:10.1145/2503210.2503296.

51 Hanan Samet. Proving the Correctness of Heuristically Optimized Code. *Communications of the ACM*, 21(7), 1978. doi:10.1145/359545.359569.

52 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594302.

53 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, 2008. doi:10.1007/978-3-540-71067-7_6.

54 Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *International Symposium on Formal Methods (FM)*, 2015. doi:10.1007/978-3-319-19249-9_33.

55 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The Verified CakeML Compiler Backend. *Journal of Functional Programming*, 29, 2019. doi:10.1017/S0956796818000229.

56 Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Munoz. An Abstract Interpretation Framework for the Round-off Error Analysis of Floating-Point Programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2018. doi:10.1007/978-3-319-73721-8_24.

57 Linus Torvalds. What is acceptable for -ffast-math?, 2001. URL: https://gcc.gnu.org/legacy-ml/gcc/2001-07/msg02150.html.

58 Vadim Zaliva. *HELIX: From Math to Verified Code*. PhD thesis, Carnegie Mellon University, 2020.

59 Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103656.2103709.

Elementary Type Inference

Jinxu Zhao ✉
Department of Computer Science, The University of Hong Kong, China

Bruno C. d. S. Oliveira ✉
Department of Computer Science, The University of Hong Kong, China

Abstract

Languages with polymorphic type systems are made convenient to use by employing type inference to avoid redundant type information. Unfortunately, features such as impredicative types and subtyping make complete type inference very challenging or impossible to realize.

This paper presents a form of partial type inference called *elementary type inference*. Elementary type inference adopts the idea of inferring only monotypes from past work on predicative higher-ranked polymorphism. This idea is extended with the addition of explicit type applications that work for any polytypes. Thus easy (predicative) instantiations can be inferred, while all other (impredicative) instantiations are always possible with explicit type applications without any compromise in expressiveness. Our target is a System F extension with top and bottom types, similar to the language employed by Pierce and Turner in their seminal work on local type inference. We show a sound, complete and decidable type system for a calculus called $F^e_{<:}$, that targets that extension of System F. A key design choice in $F^e_{<:}$ is to consider top and bottom types as polytypes only. An important technical challenge is that the combination of predicative implicit instantiation and impredicative explicit type applications, in the presence of standard subtyping rules, is non-trivial. Without some restrictions, key properties, such as subsumption and stability of type substitution lemmas, break. To address this problem we introduce a variant of polymorphic subtyping called *stable subtyping* with some mild restrictions on implicit instantiation. All the results are mechanically formalized in the Abella theorem prover.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Type Inference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.2

Related Version *Full Version*: `https://github.com/JimmyZJX/ElementaryTypeInference/blob/main/paper_extended.pdf`

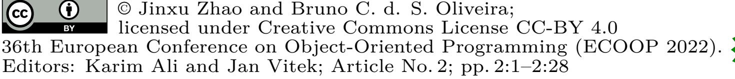
Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.5`

Funding The research is supported by the Type Inference for Complex Type Systems collaboration project (TC20210422012) between Huawei, and the University of Hong Kong and Hong Kong Research Grants Council projects number 17209520 and 17209821.

Acknowledgements We are grateful to the anonymous reviewers for their valuable comments which helped to improve the presentation of this work. We also thank Chen Cui for creating the implementation for our prototype.

1 Introduction

Many programming languages, such as Java, C#, Scala or TypeScript (among others) have type systems with parametric polymorphism, subtyping and first-class functions. For convenience, some form of type inference is desirable in those languages. Type inference avoids code being cluttered with redundant type annotations, as well as explicit type instantiations of polymorphic functions. For instance, in Java, we can write code such as:



© Jinxu Zhao and Bruno C. d. S. Oliveira;
licensed under Creative Commons License CC-BY 4.0

```
List<String> numbers = Arrays.asList("1", "2", "3", "4", "5", "6");
List<Integer> even = numbers.stream()
                            .map(s -> Integer.valueOf(s))
                            .filter(number -> number % 2 == 0)
                            .collect(Collectors.toList());
```

This code processes a list of strings representing numbers, converts the strings into numbers, and filters the even numbers. Thus the list `even` is `[2,4,6]`. The code is made practical by a form of *local type inference* [27, 21, 4, 28], which helps in two ways. Firstly, local type inference enables *synthesis of type arguments* in many cases. That is, applications of generic methods (i.e. methods parametrized by some types) will *automatically* infer the type arguments. An example of type argument synthesis is the method call `map(s -> Integer.valueOf(s))`. The polymorphic `map` function allows taking a function that converts values in the list with some type `A` into values of some other type `B`, thus producing a list with type `List<B>`. In the code above, `B` is `Integer`, and such instantiation is implicitly done by the compiler. Secondly, local type inference employs *bidirectional type-checking* [27, 10] to propagate known type-information. For example, in the snippet above the type of `numbers` is `List<String>`, therefore in `map(s -> Integer.valueOf(s))` we can deduce that `s` is of type `String`.

Without local type inference, the code above would need to explicitly provide the types for instantiation and the types of arguments for the lambda functions, making the code cluttered with type annotations. Nonetheless, some instantiations cannot be automatically inferred. For such cases, languages like Java support *explicit type applications* as well. For instance, we can write an alternative invocation of the `map` function with explicit type application as:

```
   .<Integer>map(s -> Integer.valueOf(s))
```

In their original work on local type inference, Pierce and Turner [27] considered a System F language extended with subtyping and a top and a bottom type. Such a language captures most of the essential features of interest to write code such as the above. In particular, their System F variant captures essential forms of subtyping by including top and bottom types, supports both implicit instantiation and explicit type application, and employs bidirectional type-checking for inferring types in simple cases with lambdas as arguments.

Local type inference is a pragmatic approach. It is not aimed to provide the same degree of type inference that is possible in languages with Hindley-Milner type inference [16, 19, 5]. Instead, as Pierce and Turner state, the goal is to "*exchange complete type inference for simpler methods that work well in the presence of more powerful type-theoretic features such as subtyping and impredicative polymorphism*". In local type inference, the main design decision that is made to simplify type inference is to avoid forms of global type inference that employ long-distance constraints such as unification variables.

The issues caused by both subtyping and impredicative polymorphism for more ambitious global type inference methods are well-known. There are several important undecidability results that are relevant. For instance, for System F, full type inference is undecidable [38] and an *impredicative* polymorphic subtyping relation for System F is undecidable as well [35, 3]. For a type system with top and bottom types and type variables, subtyping can also easily run into undecidable problems [34]. Despite those problems, several different approaches and restrictions have been developed for allowing type inference for: predicative versions of System F [25, 8, 9, 20, 39]; impredicative versions of System F [17, 18, 36, 33, 32, 14]; as well as several Hindley-Milner extensions with subtyping [11, 30, 6, 23].

Perhaps surprisingly, follow-up research on local type inference and type inference for type systems that include higher-ranked polymorphism, impredicativity and subtyping (with top and bottom types) has been relatively limited. In contrast, closely related global type inference approaches with higher-ranked polymorphism (HRP) for System-F-like languages

have seen quite a bit of development. Such HRP approaches [8, 17, 18, 36, 25, 33, 20] extend the classic Hindley-Milner type system, removing the restriction of top-level (let) polymorphism only. Both local type inference and HRP type inference techniques allow *synthesis of type arguments* and use type annotations to aid inference. The main difference is that HRP type inference targets a System-F-like language without subtyping, whereas the local type inference targets a System-F-like language with subtyping. Another difference is that most HRP techniques support implicit instantiation only, although there is some work on supporting visible (or explicit) type applications as well in HRP approaches [13, 32].

In this paper, we propose a form of partial type inference called *elementary type inference*. Like local type inference, we aim at having a pragmatic approach. We are willing to sacrifice some power in terms of what can be inferred, in exchange for an approach that can deal with both subtyping in the presence of top and bottom types, as well as impredicative polymorphism. Like local type inference we support synthesis of type arguments and bi-directional type-checking, but do not support Hindley-Milner style *generalization*. Unlike local type inference, our type inference is global and employs long-distance constraints. We build on recent developments in predicative type inference with HRP for System-F-like languages. The philosophy in elementary type inference is to infer only *monotypes*, but also include an explicit type application construct that can be used to instantiate *any types* (or polytypes). In other words, many programs where instantiations are monotypes can still benefit from type inference, while no expressive power is sacrificed. We can always resort to explicit type application for dealing with general polytypes.

We present a calculus with elementary type inference, called $F^e_{<:}$, that can encode all terms in a variant of System F with subtyping. The type system of $F^e_{<:}$ is a variant of the Dunfield and Krishnaswami [8] (DK) type system, extended with top and bottom types and impredicative explicit type application. The algorithmic formulation of $F^e_{<:}$ is based on the worklist algorithm by Zhao et al. [39]. We have a prototype implementation, as well as a full mechanical formalization (including results such as soundness, completeness and decidability) in the Abella theorem prover [15]. A key design choice in $F^e_{<:}$ is to consider top and bottom types as polytypes only. In other words, we avoid guessing types for implicit instantiation that use top and bottom types. This design choice avoids many of the technical challenges that would otherwise occur in the inference of terms with top and bottom types.

A key technical challenge is that the combination of predicative implicit instantiation and impredicative explicit type applications is problematic. Without some restrictions, important properties, such as subsumption and type substitution lemmas, break. To address this problem we introduce a novel polymorphic subtyping relation called *stable subtyping*. Stable subtyping has some mild restrictions, compared to the well-known Odersky and Läufer [20] formulation, but accounts for top and bottom types and impredicative instantiations. In essence, due to the presence of explicit impredicative type applications, out-of-order implicit instantiation and unused type variables are forbidden.

In summary, the contributions of this work are:

- **Elementary type inference:** A form of partial type inference that combines predicative implicit instantiation with impredicative explicit type application, in the presence of conventional subtyping rules and top and bottom types.
- **The $F^e_{<:}$ calculus:** We present a syntax-directed specification and an algorithmic version of the $F^e_{<:}$ calculus. We show that the algorithmic version is sound and complete to the specification, and its type system is also decidable. Furthermore, $F^e_{<:}$ is type-safe and complete with respect to a variant of System F with subtyping and top and bottom types.

- **Stable subtyping:** A new form of polymorphic subtyping, based on the well-known polymorphic subtyping relation by Odersky and Läufer [20], but with some restrictions. The restrictions are needed to ensure important properties such as subsumption and stability of type substitutions in the presence of impredicative type applications.
- **Implementation and mechanical formalization.** All the calculi and proofs presented in this paper are mechanically formalized in the Abella theorem prover. The formalization, an implementation and the extended version of the paper can be found in: `https://github.com/JimmyZJX/ElementaryTypeInference`

2 Overview

We start with a background on higher-ranked polymorphic (HRP) type inference and the declarative type system by Dunfield and Krishnaswami [8]. Then we discuss the challenges of extending such HRP systems with explicit type applications and top and bottom types. Finally, we illustrate the key ideas in our work to address those challenges.

2.1 Background: Higher-Ranked Type Inference and Type Applications

Our work builds on prior work on HRP type inference. In HRP type systems, universal quantification can appear in arbitrary positions in types. This lifts a restriction of Hindley-Milner type inference [16, 19, 5], where universal quantification can only appear at the top level. To introduce HRP type inference we will use examples in GHC 8, whose type inference algorithm is closely based on the work by Eisenberg et al. [13] on visible type application. The use of GHC 8 will later be helpful to illustrate some challenges of combining HRP type inference with explicit type applications and standard subtyping rules.[1]

A canonical example of an expression with an arbitrary higher-ranked type is:

```
hpoly = \(f :: forall a. a -> a) -> (f 1, f 'c')
```

The type of this function is `(forall a. a -> a) -> (Int, Char)`. A type annotation helps the type system to infer a type. In general, HRP type systems require some type annotations. In many such systems, it is enough to provide type annotations for polymorphic arguments (such as above). The use of some type annotations means that type inference is partial.

Predicativity, Polymorphic Subtyping and Explicit Type Applications. In predicative type systems, universally quantified types can only be instantiated with monotypes, which are types that do not contain universal quantifiers. For instance, the following definition of `f`:

```
f :: (forall a. Int -> a -> Int) -> Bool -> Int
f k = k 3
```

illustrates a higher-ranked function, where the argument `k` is polymorphic. In the body, implicit instantiation is used when applying `k` to an argument. In that application, the type argument of `k` is left implicit and is instantiated automatically with the monotype `Bool`.

The polymorphic subtyping relation used in GHC 8 (based on Peyton Jones et al.'s work [25]) and also by DK's type system allows implicit instantiation of type arguments in polymorphic types, which follows Hindley-Milner. This allows us, for instance, to define a function `h`, with a different but compatible type with `f`:

```
h :: (forall b a. b -> a -> b) -> Bool -> Int
h k = f k
```

[1] Type-checked with the GHC extensions: `RankNTypes`, `TypeApplications` and `ScopedTypeVariables`.

Notice that in `h`, one more universal variable is added, generalizing the argument type compared to `f`. However, since subtyping of functions is contravariant on the input types, the type of `h` is *less* general (or a supertype) of the type of `f`.

Another alternative to support instantiation is to employ an explicit type application. For example, function `g` illustrates explicit type applications in GHC Haskell:

```
g :: (forall a. Int -> a -> Int) -> Bool -> Int
g k = k @Bool 3
```

The function `g` has the same type as `f` but explicitly instantiates the type arguments of the argument `k`. The notation $e\ @\tau$ (for instance `k @Bool` in the definition of `g` above) denotes explicit type applications in Haskell. We will also adopt a similar notation in this paper.

2.2 Background: The Dunfield and Krishnaswami Type System

Our declarative type system can be viewed as a variant of the DK type system. The DK type system is predicative and supports implicit instantiation only. We review the original type system first, before proceeding with the presentation of our work.

Syntax. The syntax of DK's declarative system is shown at the top of Figure 1. A declarative type A is either the unit type 1, a type variable a, a universal quantification $\forall a.\ A$ or a function type $A \to B$. Nested universal quantifiers are allowed for types, but monotypes τ do not have any universal quantifier. Terms include a unit term $()$, variables x, lambda-functions $\lambda x.\ e$, applications $e_1\ e_2$ and annotations $(e : A)$. Contexts Ψ are sequences of type variable declarations and term variables with their types declared $x : A$.

Declarative Subtyping. The middle of Figure 1 shows DK's declarative subtyping judgment $\Psi \vdash A \leq B$, which was adopted from Odersky and Läufer [20]. This judgment compares the degree of polymorphism between types A and B in DK's implicitly polymorphic type system. If A can always be instantiated to match any instantiation of B, then A is "at least as polymorphic as" B. We also say that A is "more polymorphic than" B and write $A \leq B$. Subtyping rules $\leq$Var, $\leq$Unit and $\leq\to$ handle simple cases that do not involve universal quantifiers. The subtyping rule for function types $\leq\to$ is standard, being covariant on the return type and contravariant on the argument type. Rule $\leq\forall$R states that if A is more general than $\forall b.\ B$, then A must instantiate to $[\tau/b]B$ for every τ. The type variable b we introduced in the premise is implicitly fresh. We use this convention throughout the whole paper. The most interesting rule is $\leq\forall$L, which is where implicit instantiation can happen. If some instantiation of $\forall a.\ A$, $[\tau/a]A$, is a subtype of B, then $\forall a.\ A \leq B$. Only monotypes τ can be used to instantiate a, which is *guessed* in this declarative rule.

Declarative Typing. The bidirectional type system, shown at the bottom of Figure 1, has three judgments. The checking judgment $\Psi \vdash e \Leftarrow A$ checks expression e against the type A in the context Ψ. The synthesis judgment $\Psi \vdash e \Rightarrow A$ synthesizes the type A of expression e in the context Ψ. The application judgment $\Psi \vdash A \bullet e \Rrightarrow C$ synthesizes the type C of the application of a function of type A (which could be polymorphic) to the argument e.

Many rules are standard bidirectional type-checking rules [10], so we focus only on the more interesting and non-standard rules. Checking an expression e against a polymorphic type $\forall a.\ A$ in the context Ψ (rule D$\forall$I) succeeds if e checks against A in the extended context (Ψ, a). The subsumption rule DSub calls the subtyping relation, and changes the mode from checking to synthesis: if e synthesizes type A and $A \leq B$, then e checks against B. Besides a standard checking rule (D$\to$I) for lambda abstractions, rule D$\to$I$\Rightarrow$ synthesizes monotypes

Syntax

Type variables	a, b		
Types	A, B, C	$::=$	$1 \mid a \mid \forall a.\ A \mid A \to B$
Monotypes	τ, σ	$::=$	$1 \mid a \mid \tau \to \sigma$
Expressions	e	$::=$	$x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid (e : A)$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, x : A$

$\boxed{\Psi \vdash A \le B}$ Declarative subtyping

$$\frac{a \in \Psi}{\Psi \vdash a \le a}\ \le\texttt{Var} \qquad \frac{}{\Psi \vdash 1 \le 1}\ \le\texttt{Unit} \qquad \frac{\Psi \vdash B_1 \le A_1 \quad \Psi \vdash A_2 \le B_2}{\Psi \vdash A_1 \to A_2 \le B_1 \to B_2}\ \le\to$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \le B}{\Psi \vdash \forall a.\ A \le B}\ \le\forall\texttt{L} \qquad \frac{\Psi, b \vdash A \le B}{\Psi \vdash A \le \forall b.\ B}\ \le\forall\texttt{R}$$

$\boxed{\Psi \vdash e \Leftarrow A}$ e checks against input type A.

$\boxed{\Psi \vdash e \Rightarrow A}$ e synthesizes output type A.

$\boxed{\Psi \vdash A \bullet e \Rrightarrow C}$ Applying a function of type A to e synthesizes type C.

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \texttt{DVar} \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \le B}{\Psi \vdash e \Leftarrow B}\ \texttt{DSub} \qquad \frac{\Psi \vdash A \quad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \texttt{DAnno}$$

$$\frac{}{\Psi \vdash () \Rightarrow 1}\ \texttt{D1I}\Rightarrow \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \bullet e \Rrightarrow C}{\Psi \vdash \forall a.\ A \bullet e \Rrightarrow C}\ \texttt{D}\forall\texttt{App} \qquad \frac{\Psi, a \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall a.\ A}\ \texttt{D}\forall\texttt{I}$$

$$\frac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B}\ \texttt{D}\to\texttt{I} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \to C \bullet e \Rrightarrow C}\ \texttt{D}\to\texttt{App}$$

$$\frac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau}\ \texttt{D}\to\texttt{I}\Rightarrow \qquad \frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rrightarrow C}{\Psi \vdash e_1\ e_2 \Rightarrow C}\ \texttt{D}\to\texttt{E}$$

Figure 1 The Dunfield and Krishnaswami Type System.

$\sigma \to \tau$. Application $e_1\ e_2$ is handled by rule `D→E`, which first synthesizes the type A of the function e_1. If A is a function type $B \to C$, then rule `D→App` is applied. The synthesized type of function e_1 can also be polymorphic, of the form $\forall a.\ A$. In that case, we instantiate A to $[\tau/a]A$ with a monotype τ using the rule `D∀App`. If $[\tau/a]A$ is a function type, rule `D→App` is used; if $[\tau/a]A$ is another universal quantified type, rule `D→I⇒` is recursively applied.

2.3 The Challenges of Explicit Type Applications

While explicit type applications seem like a natural extension to type systems with implicit instantiation, the combination can break some important properties and make programs less robust to refactoring or inlining. In particular, in many existing type systems with implicit instantiation, the order of type arguments being instantiated does not matter, whereas with explicit instantiation the order does matter (at least if arguments are positional). The design proposed by Eisenberg et al. [13] notices this difference, and distinguishes between specified and generalized type quantification. All the GHC examples that we show in this

paper employ specified type quantification, where the programmer explicitly writes the type signature, and the order of type arguments is relevant. However, there are other subtler points in the design of a subtyping relation for specified type quantification that make it hard to get certain expected properties in a language. We illustrate some concrete issues next with GHC 8 and Eisenberg et al.'s design. We remark, however, that some other issues with GHC 8's approach have already been identified and GHC 9 adopts a different approach that avoids the issues described here. A detailed discussion follows in Section 7.

Explicit Type Applications, Subsumption and Equational Reasoning. The design by Eisenberg et al. is based on bidirectional type-checking and supports a standard subsumption rule, similar to the rule `DSub` in DK's type system. Moreover, the subtyping relation employed in that design is essentially an extension of that in DK's type system. The examples that we show next involve subtyping relations that are valid in both DK's type system, as well as the subtyping relation employed by Eisenberg et al. [13]. An important lemma that holds for the DK type system is the checking subsumption lemma:

▶ **Lemma 1** (Checking Subsumption). *If* $\Psi \vdash e \Leftarrow A$ *and* $\Psi \vdash A \leq B$ *then* $\Psi \vdash e \Leftarrow B$

This lemma is quite similar to the subsumption rule. The difference is that the premise ($\Psi \vdash e \Leftarrow A$) is in checking mode, instead of synthesis mode. The lemma states that we can always change the type of an expression being checked to a supertype. A practical consequence of this lemma is that changing type annotations of an expression to a supertype is always possible, which is something that programmers would expect. For example, we could change the type annotation of `f` in Section 2.1 to that of `h` and `f` would still type-check.

In contrast to DK's type system, the work by Eisenberg et al. [13] and GHC 8 does not have the checking subsumption property. We believe that the lack of this property is undesirable, especially for a language that promotes equational reasoning like Haskell. To illustrate why the property is important, consider the Haskell functions:

```
h2 :: (forall b a. b -> a -> b) -> Bool -> Int
h2 k = g k -- type checks!

h3 :: (forall b a. b -> a -> b) -> Bool -> Int
h3 k = k @Bool 3 -- rejected!
```

Recall the examples from Section 2.1. Function `h` is defined with a simple call to `f`, although it has a different type. Here the new function `h2` has the same type as `h`, which is a supertype of the type of `f` and `g`. However, if we try to replace the call `g k` by its definition in `h2`, we get the definition `h3`, which no longer type-checks. There are two important issues to notice here. Firstly, replacing equals by equals (or inlining) results in a program that does not type-check! The problem is that now the type argument of `k` in `h3` instantiates the *wrong* type variable. Now `k` has two type arguments, and the second type argument corresponds to the first type argument of the type of the argument `k` in `g`. Because of this, the type of `k` in `h3` is not compatible with the use of `k` in the body. Secondly, the example shows that higher-ranked type arguments that use explicit type applications can break the checking subsumption property. There are some valid supertypes of functions that, if used instead of the original type, will result in an ill-typed program. Even though the type of `h2` is a supertype of the type of `g`, we cannot use that supertype to type-check `g`.

Explicit Type Applications in DK's Type System. A naive extension of the DK type system with explicit type applications would be to add the rule:

$$\frac{\Psi \vdash e \Rightarrow \forall a.\ A}{\Psi \vdash e\ @B \Rightarrow [B/a]A}\ \texttt{DTypeApp}\forall$$

This rule first synthesizes a polymorphic type $\forall a.\ A$ from e, and then outputs its instantiation $[B/a]A$ for the explicit type application $e\ @B$. Unfortunately, this rule breaks checking subsumption as well, for very similar reasons to those in the GHC examples. It is easy to port the previous GHC counter-examples to this extension of DK's type system, and get similar issues to those that we have just described.

▶ **Example 2.** Now we look at a different example that illustrates the general problem of having order-irrelevance of universally quantified type variables. Suppose that we have $e = \lambda x.\ x\ @\text{Int}\ 3\ \text{True}$, $A = (\forall a.\ \forall b.\ a \to b \to a) \to \text{Int}$, and $B = (\forall b.\ \forall a.\ a \to b \to a) \to \text{Int}$. Both conditions of the checking subsumption lemma

$$\lambda x.\ x\ @\text{Int}\ 3\ \text{True} \Leftarrow (\forall a.\ \forall b.\ a \to b \to a) \to \text{Int}$$
$$(\forall a.\ \forall b.\ a \to b \to a) \to \text{Int} \le (\forall b.\ \forall a.\ a \to b \to a) \to \text{Int}$$

hold, yet the conclusion $\big(\lambda x.\ x\ @\text{Int}\ 3\ \text{True} \Leftarrow (\forall b.\ \forall a.\ a \to b \to a) \to \text{Int}\big)$ does not hold: the type application instantiates the wrong type variable, causing the type system to reject it. The types A and B here differ only in the order of polymorphic type variables. The order-sensitive explicit type application is not compatible with order-irrelevant subtyping relations, such as the one in DK's type system, breaking the checking subsumption lemma.

Impredicative Type Applications and Stability of Subtyping. So far, we have illustrated problems that involve only monotypes (and predicative instantiation). If impredicative type applications are supported as well, then this brings another class of problems.

▶ **Example 3.** Consider $e = \lambda x.\ x\ @C$, $A = (\forall a.\ a \to a) \to (C \to C)$, and $B = (\forall a.\ \forall b.\ b \to a) \to (C \to C)$. Here we assume $C = (\forall a.\ a \to a) \to \text{Int}$, but other polymorphic types could be used as well. Both conditions of the checking subsumption lemma:

$$\lambda x.\ x\ @C \Leftarrow (\forall a.\ a \to a) \to (C \to C)$$
$$(\forall a.\ a \to a) \to (C \to C) \le (\forall a.\ \forall b.\ b \to a) \to (C \to C)$$

hold, but not the conclusion $\lambda x.\ x\ @C \Leftarrow (\forall a.\ \forall b.\ b \to a) \to (C \to C)$. When type checking the condition $\lambda x.\ x\ @C \Leftarrow (\forall a.\ a \to a) \to (C \to C)$, the type application $@C$ is properly applied, instantiating the universal variable a on the type of x $(\forall a.\ a \to a)$ to C. However, when we have the conclusion $\lambda x.\ x\ @C \Leftarrow (\forall a.\ \forall b.\ b \to a) \to (C \to C)$, the type application $x\ @C$ only instantiates the type argument a, but not the type argument b. Thus the type inferred for the application $x\ @C$ is $\forall b.\ b \to C$. Unfortunately, the predicative subtyping relation rejects $\forall b.\ b \to C \le C \to C$ when C is a polymorphic type like $(\forall a.\ a \to a) \to \text{Int}$. Thus, the conclusion $\lambda x.\ x\ @C \Leftarrow (\forall a.\ \forall b.\ b \to a) \to (C \to C)$ fails to type check.

While this problem is also an instance of checking subsumption not holding, the reason why this example fails is different from the previous examples. The crux of the problem here is that some instantiations with polytypes break the polymorphic subtyping relation after instantiation. More concretely, we would like the following property to hold:

▶ **Corollary 4** (Stability of Subtyping). *If* $\Psi \vdash \forall a.\ A \leq \forall a.\ B$ *then* $\Psi \vdash [C/a]A \leq [C/a]B$ *holds for any well-formed* C $(\Psi \vdash C)$*.*

but this property does not hold in general for polytypes in DK's type system. DK's type system has a similar property, but only for monotypes. When impredicative type applications are present, we would like to have a more general property that holds for polytypes as well.

Problem with Unused Variables. Furthermore, unused type variables in universal quantifiers are problematic, since they can also break stability of subtyping. $\forall a.\ \forall b.\ a \leq \forall a.\ a$ is accepted by most subtyping relations that support implicit instantiation, including DK and HM. If we instantiate both sides of the subtyping judgment with the polytype $C := \forall c.\ c \to c$, the judgment becomes $\forall b.\ \forall c.\ c \to c \leq \forall c.\ c \to c$, which is then problematic, because a second instantiation with $C' := \forall d.\ d \to d$ would give $\forall c.\ c \to c \leq (\forall d.\ d \to d) \to (\forall d.\ d \to d)$, which is rejected by predicative systems. Here the problem is not caused by a permutation or extra type variables in polytypes. Instead, an unused variable leads to such problem.

2.4 The Problems with Subtyping with Top and Bottom Types

Besides the problems with explicit type applications, the presence of top and bottom types raises its own class of issues.

Polymorphic Subtyping with Top and Bottom Types. To support top and bottom types in DK's type system, a first idea is to introduce those types, with their standard subtyping rules and consider them to be monotypes, leading to the following syntax for monotypes:

$$\tau ::= 1 \mid \bot \mid \top \mid \tau \to \tau \mid a$$

Unfortunately, it is known that the subtyping for a language with type variables, such as the above, can quickly become undecidable [34]. To illustrate some of the issues, consider the subtyping judgment $\forall a.\ (a \to a \to 1) \to 1 \leq (\forall b.\ b \to b) \to 1$, which reduces to the following problem: find (predicative) instantiations for $\widehat{\alpha}$ and $\widehat{\beta}$, satisfying $\widehat{\alpha} \leq \widehat{\beta}$ and $\widehat{\beta} \leq \widehat{\alpha} \to 1$. Such a problem has infinitely many solutions, where there is no best one. If we assume that existential variables $\widehat{\alpha}$ and $\widehat{\beta}$ are instantiated to the same type, there are infinite solutions:

$$\widehat{\alpha} = \widehat{\beta} = \bot \mid \top \to 1 \mid \top \to \bot \mid (\bot \to 1) \to 1 \mid \cdots$$

Additionally, $\widehat{\alpha} = \widehat{\beta}$ is not the most general unification for the judgment $\widehat{\alpha} \leq \widehat{\beta}$. Assignments like $\widehat{\alpha} = \widehat{\beta}, \widehat{\beta} = \widehat{\beta} \to 1$ also validate the subtyping judgments.

Inference of $\top$ and $\bot$ Types can Mask Type Errors errors. Consider the following expression:

$$\lambda f.\ f + f\ 1$$

Such expression can be typed with the type $\bot \to \mathrm{Int}$. Since the input parameter f, having type $\bot$, can be converted to either an integer or to a function of type $\mathrm{Int} \to \mathrm{Int}$. However, inferring such a type is hardly useful in practice. Instead, the programmer might have immediately realised the bug (perhaps the argument for the first call to f is missing) if the type inference algorithm rejects the lambda expression, rather than after it has inferred a type with $\bot$. By constraining the type inference algorithm to infer types free from $\top$ and $\bot$, such type is no longer be inferred, and thus the bug is reported when defining the function.

It is mentioning that, in languages like Scala, there are several reports of issues arising because $\top$ and $\bot$ (respectively *Any* and *Nothing* in Scala) can be inferred[2]. The Scala compiler even has a flag `-Xlint:infer-any` that is used to warn whenever *Any* is inferred.

2.5 Our Solution

The form of type-inference available in $F^e_{<:}$ is inspired by current approaches employed in predicative higher-ranked type inference [25, 8, 9, 20, 39]. In terms of restrictions, $F^e_{<:}$ does not have generalization (similarly to the DK type system), and there are some restrictions that weaken the expressive power of the polymorphic subtyping relation. However, those restrictions mostly affect higher-ranked programs, and for many other Hindley-Milner style programs (with polymorphic annotations) there should be no impact from those restrictions. In terms of innovations the type system of $F^e_{<:}$ supports top and bottom types and explicit impredicative type applications. Moreover, implicit instantiation and explicit instantiation interoperate well and have important properties, such as checking subsumption and stability of subtyping. Next we show some examples that run in our implementation and illustrate the capabilities of the $F^e_{<:}$ type system. We note that our implementation contains some extra features that enable us to present more interesting examples. These features include recursive let expressions, (polymorphic) lists and case expressions on lists.

Rank-1 Polymorphism. We start with first-order polymorphism, which is the kind of polymorphism supported in Hindley-Milner. We can define the `map` function in $F^e_{<:}$ as follows:

```
1 let map :: forall a b. (a -> b) -> [a] -> [b]
2     = \f -> \xs -> case xs of [] -> []; (x:xs) -> f x : map f xs
```

This definition is similar to a definition in a language with Hindley-Milner, except that we must explicitly provide the type of the `map` function. An explicit type is optional in languages like Haskell or ML, but must be provided in $F^e_{<:}$ for polymorphic functions like `map`. Like in Hindley-Milner, when writing the body of the function we do not need to use type binders and the recursive call implicitly instantiates the types `a` and `b`. We can use `map` conventionally, as in an HM language like Haskell or ML:

```
1 map (\x -> x + 1) [1,2,3]
```

or use explicit type applications if necessary or desired. For example:

```
1 map @Int @Top (\x -> x :: Top) [1,2,3]
```

which uses explicit type applications to instantiate `b` with the `Top` type. Since the body of the function argument uses a type annotation (`x :: Top`) to return a `Top` type, writing `map (\x -> x :: Top) [1,2,3]` would fail to type-check. The problem is that `Top` is not a monotype in our system and cannot be inferred during implicit instantiation.

While there is no generalization, we can still infer monotypes for lambdas. Therefore, the following is allowed:

```
1 let succ = \x -> x + 1
```

The type inferred for `succ` is the monotype `Int -> Int`. However writing `let id = \x -> x`, without an explicit annotation for `id` would fail, since generalization would be necessary to infer a polymorphic type for the identity function.

[2] See for instance: `https://riptutorial.com/scala/example/21134/preventing-inferring-nothing`.

Higher-Ranked Polymorphism. With explicit type applications, it becomes possible to perform impredicative instantiations. A simple example of this is applying the identity function to itself. In $F^e_{<:}$ we can write:

```
let id :: forall a. a -> a = \x -> x in id @(forall a. a -> a) id
```

In this case the type used to explicitly instantiate the identity function is a polymorphic type. Implicit instantiation is not possible here, since the type argument is not a monotype. Another example of impredicative instantiation is:

```
let plist :: [forall a. a -> a] = [\z -> z, \z -> z]
in map @(forall a. a -> a) (\f -> f 1) plist
```

where the `map` function takes a list with *polymorphic* functions of type `forall a. a -> a`. The function argument (`\f -> f 1`) of `map` applies a polymorphic function to `1`.

Restrictions for Higher-Ranked Polymorphism. The GHC 8 definitions in Section 2.1 for `f` and `g` also type-check in $F^e_{<:}$:

```
let f :: (forall a. Int -> a -> Int) -> Bool -> Int = \k -> k 3
let g :: (forall a. Int -> a -> Int) -> Bool -> Int = \k -> k @Bool 3
```

However, the definition of `h` fails to type-check:

```
let h :: (forall b. forall a. b -> a -> b) -> Bool -> Int = \k -> f k  -- fails!
```

This definition is rejected because $\forall b.\ \forall a.\ b \to a \to b \leq \forall a.\ \mathrm{Int} \to a \to \mathrm{Int}$ does not hold in $F^e_{<:}$. Our polymorphic subtyping relation does not consider the type of `k` to be a subtype of the expected argument for `f`. By preventing examples such as this one we avoid the issues discussed in Section 2.3 and we retain the checking subsumption property.

We can also type-check the expression used in our Example 2:

```
\x -> x @((forall a. a -> a) -> Int)
```

with a type annotation:

```
(forall a. a -> a) -> ((forall a. a -> a) -> Int) -> ((forall a. a -> a) -> Int)
```

owever, using the type annotation

```
(forall a. forall b. b -> a) ->
  ((forall a. a -> a) -> Int) -> ((forall a. a -> a) -> Int)
```

will fail because the type in the first annotation is not a subtype of the type in the second annotation in our polymorphic subtyping relation: the subtyping statement $\forall a.\ \forall b.\ b \to a \leq \forall a.\ a \to a$ does not hold. In the Odersky and Läufer relation, such subtyping statement holds. As discussed in Section 2.3, accepting such subtyping statements is problematic when impredicative instantiation is allowed, since we lose the stability of subtyping property.

No Inference of Top and Bottom. Finally the type system of $F^e_{<:}$ rejects:

```
let strange = \f -> f + f 1   -- fails!
```

As discussed in Section 2.4, this definition could be type-checked if we could infer the bottom type for `f`. Since bottom is not a monotype, it cannot be infered and this definition is rejected. However, the following definition, with an explicit type annotation is allowed:

```
let strange : Bot -> Int = \f -> f + f 1
```

In other words, we avoid inferring top and bottom, which can type-check programs that are likely to have type errors. However, we can always provide annotations for definitions that can type-check with polytypes, thus allowing such definitions to type-check if desired.

$\boxed{\Psi \vdash_s A \leq B}$ Stable subtyping

$$\frac{\Psi \vdash_s A}{\Psi \vdash_s A \leq A}\leq_{\mathtt{s}}\mathtt{refl} \qquad \frac{}{\Psi \vdash_s A \leq \top}\leq_{\mathtt{s}}\top \qquad \frac{}{\Psi \vdash_s \bot \leq A}\leq_{\mathtt{s}}\bot$$

$$\frac{\forall C, \Psi \vdash_s C \implies \Psi \vdash_s [C/a]A \leq [C/a]B}{\Psi \vdash_s \forall a.\ A \leq \forall a.\ B}\leq_{\mathtt{s}}\forall \qquad \frac{\Psi \vdash_s B_1 \leq A_1 \quad \Psi \vdash_s A_2 \leq B_2}{\Psi \vdash_s A_1 \to A_2 \leq B_1 \to B_2}\leq_{\mathtt{s}}\to$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash_s [\tau/a]A \leq B \quad B \neq \forall b.B'}{\Psi \vdash_s \forall a.\ A \leq B}\leq_{\mathtt{s}}\forall\mathtt{L}$$

Figure 2 Stable Subtyping.

2.6 Key Technical Ideas

Stable Polymorphic Subtyping. We address the problems with explicit type applications via a novel notion of polymorphic subtyping, which preserves both subsumption and stability of subtyping. Stable subtyping, shown in Figure 2, is a variant from DK's subtyping with some changes. Note that we use color to highlight new rules or changes, with respect to DK's type system, here and throughout the paper. The new rule $\leq_{\mathtt{s}}\mathtt{refl}$ replaces base cases in the previous system, and rule $\leq_{\mathtt{s}}\forall$ directly expresses the expected stability property. We also forbid polymorphic types that contain unused type variables. This restriction is enforced by the well-formedness relation (see details in Section 3.1). Rule $\leq_{\mathtt{s}}\forall\mathtt{L}$ has a side condition to prevent overlapping with rule $\leq_{\mathtt{s}}\forall$ and has a less priority. In addition, we also have two new (but standard) rules for top and bottom types (rules $\leq_{\mathtt{s}}\top$ and $\leq_{\mathtt{s}}\bot$).

There are 3 main differences with respect to DK's subtyping relation. Firstly, stable subtyping does not allow instantiations out-of-order. Note that, unlike DK's type system, there is no rule that corresponds to $\leq\forall\mathtt{R}$, which removes the ability to perform such instantiations. Thus, the order of type variables becomes relevant. Secondly, types with unused type variables, such as $\forall a.1$, are not allowed. Finally, top and bottom types are supported.

A Syntax-Directed System with Subtype Variables. While rule $\leq_{\mathtt{s}}\forall$ directly captures the stability property that we want, such a rule is highly declarative. Thus, an important challenge is to find an alternative equivalent set of rules that is closer to an implementation. Our solution to the problem relies on a new sort of variables, $\tilde{a}$, called subtype variables, and the following subtyping rule, which is proven to have equal effect to $\leq_{\mathtt{s}}\forall$:

$$\frac{\Psi, \tilde{a} \vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B}{\Psi \vdash \forall a.\ A \leq \forall a.\ B}\leq\forall$$

The difference between a subtype variable $\tilde{a}$ and a conventional type variable a is that a subtype variable is *not a monotype*, therefore it cannot be instantiated with rule $\leq\forall\mathtt{L}$. Thus, a subtyping statement such as $\forall a.\ \forall b.\ b \to a \leq \forall a.\ a \to a$ does not hold, since it would require instantiation with a subtype variable, which is not allowed.

Explicit Type Applications for Polytype Instantiations. To avoid being overly restrictive, we still support polytype instantiations via explicit type applications. Thus, while some convenience afforded by more expressive formulations of type inference is lost, no expressive

power is lost. We can encode a variant of System $F_{<:}$ (without bounded quantification) trivially using explicit type applications. We formally verify the soundness and completeness theorems in our Abella formalization and present the results in the extended version.

Summary. With stable subtyping, Corollary 4 and the subsumption lemma hold, and the system works smoothly with the explicit type application rules in the type system. Moreover, we address the problems with top and bottom types by not considering top and bottom types as monotypes. This extends a similar idea in predicative HRP, which excludes universal types from monotypes, thus avoiding decidability issues that arise from including such types.

3 Syntax-Directed System

This section introduces a syntax-directed type system for $F^{e}_{<:}$, which serves as a specification for the algorithmic version that will be presented in Section 4. The type system can be viewed as a variant of the Dunfield and Krishnaswami [8] type system, adding explicit type applications and abstractions, as well as $\top$ and $\bot$ types. Furthermore, the type system supports impredicativity via explicit instantiations. The subtyping relation employed in this type system is equivalent to the *stable subtyping* relation introduced in Section 2.6, but it is syntax-directed and employs a special kind of type variables in the subtyping relation. Several important properties, such as the subsumption lemma, a generalized stability lemma for impredicative types, and transitivity of subtyping are proved.

3.1 Syntax and Well-Formedness

Compared to the syntax of the DK system presented in Figure 1, the syntax of $F^{e}_{<:}$:

Type variables	a, b	Subtype variables	$\tilde{a}, \tilde{b}$
Types	A, B, C	$::=$	$1 \mid a \mid \forall a.\ A \mid A \to B \mid \tilde{a} \mid \top \mid \bot$
Monotypes	τ, σ	$::=$	$1 \mid a \mid \tau \to \sigma$
Expressions	e, t	$::=$	$x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid (e : A) \mid e\ @A \mid \Lambda a.\ e : A$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, x : A \mid \Psi, \tilde{a}$

is extended in the four directions. First, types now include $\top$ and $\bot$. Second, expressions are extended with type applications ($e\ @A$). Third, type abstractions ($\Lambda a.\ e : A$), studied by Zhao et al. [39], are adopted as well, since they are useful to express programs with *scoped type variables* [24]. Note that programmers do not have to write Λ's directly. Instead, an explicit annotation $e : \forall a.\ A$ can serve as the syntactic sugar $e : \forall a.\ A \equiv \Lambda a.e : \forall a.\ A$. Finally, there is a new syntactic sort: subtype variables, $\tilde{a}$, representing type variables that are only used in subtyping and are not monotypes. It is worth mentioning that the definition for monotypes is not changed. We do not treat $\top$, $\bot$, or subtype variables as monotypes.

Well-Formedness. The well-formedness relation is mainly used to ensure the well-scopedness of binders. Additionally, we add special free variable checks to ensure that the polymorphic type $\forall a.\ A$ is indeed polymorphic in the following two rules:

$$\frac{\Psi, a \vdash A \quad a \in \mathrm{FV}(A)}{\Psi \vdash \forall a.\ A}\ \texttt{wf}_{\texttt{d}}\forall \qquad \frac{\Psi, a \vdash A \quad \Psi, a \vdash e \quad a \in \mathrm{FV}(A)}{\Psi \vdash \Lambda a.\ e : A}\ \texttt{wf}_{\texttt{d}}\texttt{tLam}$$

The motivation for the $a \in \mathrm{FV}(A)$ restriction is discussed in Section 2.3 with examples. This has an impact on the compatibility with other systems, since we can no longer express types like $\forall a.\ 1$. Yet we argue that such types are not very useful in practice; when $a \notin \mathrm{FV}(A)$, $\forall a.\ A$ is isomorphic to A in systems without explicit instantiation.

$\boxed{\Psi \vdash A \leq B \quad \hookrightarrow E}$ Syntax-directed subtyping

$$\frac{}{\Psi \vdash 1 \leq 1 \quad \hookrightarrow \lambda(x:1).\ x}\ \texttt{≤Unit} \qquad \frac{}{\Psi \vdash A \leq \top \quad \hookrightarrow \lambda(x:|A|).\ \mathrm{top}}\ \texttt{≤⊤}$$

$$\frac{}{\Psi \vdash \bot \leq A \quad \hookrightarrow \lambda(x:\forall a.\ a).\ x\ @|A|}\ \texttt{≤⊥} \qquad \frac{a \in \Psi}{\Psi \vdash a \leq a \quad \hookrightarrow \lambda(x:a).\ x}\ \texttt{≤Var}$$

$$\frac{\tilde{a} \in \Psi}{\Psi \vdash \tilde{a} \leq \tilde{a} \quad \hookrightarrow \lambda(x:a).\ x}\ \texttt{≤SVar}$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B \quad \hookrightarrow E \qquad B \neq \forall b.B' \quad B \neq \top}{\Psi \vdash \forall a.\ A \leq B \quad \hookrightarrow \lambda(x:|\forall a.\ A|).\ E\ (x\ @\tau)}\ \texttt{≤∀L}$$

$$\frac{\Psi, \tilde{a} \vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B \quad \hookrightarrow E}{\Psi \vdash \forall a.\ A \leq \forall a.\ B \quad \hookrightarrow \lambda(x:|\forall a.\ A|).\ \Lambda a.\ E\ (x\ @a)}\ \texttt{≤∀}$$

$$\frac{\Psi \vdash B_1 \leq A_1 \quad \hookrightarrow E_1 \qquad \Psi \vdash A_2 \leq B_2 \quad \hookrightarrow E_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2 \quad \hookrightarrow \lambda(f:|A_1 \to A_2|).\ \lambda(x:|B_1|).\ E_2\ (f\ (E_1\ x))}\ \texttt{≤→}$$

$\boxed{\Psi \vdash e \Leftarrow A \quad \hookrightarrow E}$ e checks against input type A.

$\boxed{\Psi \vdash e \Rightarrow A \quad \hookrightarrow E}$ e synthesizes output type A.

$\boxed{\Psi \vdash A \bullet e \Rrightarrow C \quad \hookrightarrow E_c \mid E}$ Applying a function of type A to e synthesizes type C.

$$\frac{(x:A) \in \Psi}{\Psi \vdash x \Rightarrow A \quad \hookrightarrow x}\ \texttt{DVar} \qquad \frac{\Psi \vdash e \Rightarrow A \quad \hookrightarrow E \qquad \Psi \vdash A \leq B \quad \hookrightarrow co \qquad A \neq \forall a.\ A'}{\Psi \vdash e \Leftarrow B \quad \hookrightarrow co\ E}\ \texttt{DSub}$$

$$\frac{\Psi \vdash A \qquad \Psi \vdash e \Leftarrow A \quad \hookrightarrow E}{\Psi \vdash (e:A) \Rightarrow A \quad \hookrightarrow E}\ \texttt{DAnno} \qquad \frac{\Psi \vdash \sigma \to \tau \qquad \Psi, x:\sigma \vdash e \Leftarrow \tau \quad \hookrightarrow E}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau \quad \hookrightarrow \lambda(x:\sigma).\ E}\ \texttt{D→I⇒}$$

$$\frac{\Psi, a \vdash e \Leftarrow A \quad \hookrightarrow E}{\Psi \vdash e \Leftarrow \forall a.\ A \quad \hookrightarrow \Lambda a.\ E}\ \texttt{D∀I} \qquad \frac{\Psi, x:A \vdash e \Leftarrow B \quad \hookrightarrow E}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B \quad \hookrightarrow \lambda(x:A).\ E}\ \texttt{D→I}$$

$$\frac{}{\Psi \vdash () \Rightarrow 1 \quad \hookrightarrow ()}\ \texttt{D1I⇒} \qquad \frac{\Psi \vdash e_1 \Rightarrow A \quad \hookrightarrow E_1 \qquad \Psi \vdash A \bullet e_2 \Rrightarrow C \quad \hookrightarrow E_c \mid E_2}{\Psi \vdash e_1\ e_2 \Rightarrow C \quad \hookrightarrow (E_c\ E_1)\ E_2}\ \texttt{D→E}$$

$$\frac{\Psi \vdash e}{\Psi \vdash e \Leftarrow \top \quad \hookrightarrow \mathrm{top}}\ \texttt{D⊤} \qquad \frac{\Psi \vdash e \Leftarrow A \quad \hookrightarrow E}{\Psi \vdash A \to C \bullet e \Rrightarrow C \quad \hookrightarrow \lambda(x:|A \to C|).\ x \mid E}\ \texttt{D→App}$$

$$\frac{\Psi \vdash e \Rightarrow \bot \quad \hookrightarrow E}{\Psi \vdash e\ @B \Rightarrow \bot \quad \hookrightarrow E}\ \texttt{DTA⊥}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash [\tau/a]A \bullet e \Rrightarrow C \quad \hookrightarrow E_c \mid E}{\Psi \vdash \forall a.\ A \bullet e \Rrightarrow C \quad \hookrightarrow \lambda(x:|\forall a.\ A|).\ E_c\ (x\ @\tau) \mid E}\ \texttt{D∀App}$$

$$\frac{\Psi, a \vdash e \Leftarrow A \quad \hookrightarrow E}{\Psi \vdash \Lambda a.\ e : A \Rightarrow \forall a.\ A \quad \hookrightarrow \Lambda a.\ E}\ \texttt{DSTV} \qquad \frac{\Psi \vdash e \Rightarrow \forall a.\ A \quad \hookrightarrow E}{\Psi \vdash e\ @B \Rightarrow [B/a]A \quad \hookrightarrow E\ @|B|}\ \texttt{DTA∀}$$

$$\frac{\Psi \vdash e}{\Psi \vdash \bot \bullet e \Rrightarrow \bot \quad \hookrightarrow \lambda(x:\forall a.\ a).\ x\ @(\top \to \forall a.\ a) \mid \mathrm{top}}\ \texttt{D⊥App}$$

Figure 3 Syntax-directed System.

3.2 Subtyping and Typing Rules

The top of Figure 3 shows the subtyping relation. Grayed parts are $F_{<:}$ [2] expressions, which are used to prove our soundness result with respect to $F_{<:}$, can be ignored for the moment. We refer the reader to the our extended version for the details of the $F_{<:}$ soundness result. Most rules are inherited from Odersky and Läufer (OL) [20], yet there are several differences. Rule $\leq$`SVar` is a new rule for subtype variables. This rule is just like the standard rule for type variables (rule $\leq$`Var`). Rules $\leq\top$ and $\leq\bot$ are new (but standard) rules for $\top$ and $\bot$. Rule $\leq\forall$ is also new. In this rule, two forall types are subtypes if their bodies are subtypes. Importantly, the type variables in the bodies become subtype variables and are marked as such when added to the context. Rule $\leq\forall$`L` has an two additional premises to prevent overlapping with rules $\leq\forall$ and $\leq\top$, respectively. The first condition $B \neq \forall b.\ B'$ ensures that $\leq\forall$ always has priority. The second condition $B \neq \top$ can be safely omitted without changing the expressive power, but is presented here to ensure that the system is syntax-directed.

The polymorphic subtyping relation behaves slightly differently from OL's subtyping. If we ignore $\top$ and $\bot$ types, this relation is weaker than OL. In $F^e_{<:}$, the order of polymorphic variables is important. For example, in the OL's system, the subtyping statement $\cdot \vdash \forall a.\ \forall b.\ a \to b \to a \leq \forall b.\ \forall a.\ a \to b \to a$ holds, but $F^e_{<:}$ will reduce that to $\tilde{a}, \tilde{b} \vdash \tilde{a} \to \tilde{b} \to \tilde{a} \leq \tilde{b} \to \tilde{a} \to \tilde{b}$, which does not hold. Apart from the ordering of variables, instantiation works differently for subtyping between polymorphic types. OL's subtyping relation accepts the following judgment $\cdot \vdash \forall a.\ (a \to a) \to (a \to a) \leq \forall a.\ a \to a$ but $F^e_{<:}$ does not, since $\tilde{a} \vdash (\tilde{a} \to \tilde{a}) \to (\tilde{a} \to \tilde{a}) \leq \tilde{a} \to \tilde{a}$ does not hold either.

Typing. The bottom of Figure 3 shows the type system. Compared to DK's type system, there are 4 groups of changes:

1. Rules `D`$\top$ and `D`$\bot$`App` are introduced for the $\top$ and $\bot$ types. Note that in rule `D`$\top$ we employ a relation $\Psi \vdash e$ that checks for the well-formedness of expressions. We omit the definition of $\Psi \vdash e$, but it is standard, checking whether all the free variables in e are bound in Ψ. Both rules are essential for the subsumption lemma to hold. For example, rule `D`$\bot$`App` is required to type-check the expression $(\lambda x.\ x\ ()) : \bot \to 1$, where the argument x has type $\bot$ and the application $x\ ()$ synthesizes $\bot$ according to the rule, which can then check against the 1 type by rule `DSub`.
2. Rule `DSub` now requires one side-condition to prevent overlapping with Rule `D`$\forall$`I`. In presence of explicit type applications, this condition cannot be eliminated.
3. Rules `DTA`$\bot$ and `DTA`$\forall$ infer type application expressions. If the type of e synthesizes a polymorphic type $\forall a.\ A$, then $e\ @B$ has type $[B/a]A$. Any expression of type $\bot$ will synthesize $\bot$ for any type applied.
4. Rule `DSTV` enables the scoped type variables [24]. This allows flexible control of type variables by the programmer.

Note that the `D`$\top$ is peculiar in that it allows some ill-typed terms to type-check. Such rules are often needed in bi-directional type systems with top types to enable properties such as checking subsumption. For instance a similar rule is employed by Dunfield [7]. Since the top type is the supertype of all types, all well-typed expressions should be able to type-check under the top type as well. For example, we should be able to change the type annotation in function $\lambda x.x : \mathrm{Int} \to \mathrm{Int}$ to $\lambda x.x : \top$. However, there is not enough type information to type-check the body of the later lambda. Nevertheless, we do not need to evaluate expressions with a top type, since no information can be extracted from such type, and the elaboration to $F_{<:}$ results directly in the *top* value for such expression, preserving type-safety.

3.3 Metatheory

The type system has several desirable properties, including subsumption and a stability of type substitutions lemma in subtyping.

Reflexivity and Transitivity. Firstly, our subtyping relation is reflexive and transitive.

▶ **Lemma 5** (Subtyping Reflexivity). *If* $\Psi \vdash A$ *then* $\Psi \vdash A \leq A$.

▶ **Lemma 6** (Subtyping Transitivity). *If* $\Psi \vdash A \leq B$ *and* $\Psi \vdash B \leq C$ *then* $\Psi \vdash A \leq C$.

Equivalence to Stable Subtyping and Stability. Secondly, the syntax-directed formulation of subtyping is sound and complete with respect to the stable subtyping relation in Section 2.6. Subtype variables are used to provide an alternative formulation of the $\leq\forall$ rule, bringing subtyping closer to an algorithm. Nonetheless, syntax-directed subtyping still guesses monotypes, thus it is not algorithmic.

▶ **Theorem 7** (Soundness w.r.t stable subtyping). *If* $\Psi \vdash A \leq B$ *then* $\Psi \vdash_s A \leq B$.

▶ **Theorem 8** (Completeness w.r.t stable subtyping). *Given* $\Psi \vdash A$ *and* $\Psi \vdash B$*, if* $\Psi \vdash_s A \leq B$ *then* $\Psi \vdash A \leq B$.

The proof works by generalizing instantiations for subtype variables and existential variables (which represents monotypes to be guessed). We refer to the extended version for details. A related property of the subtyping relation is stability. The following lemma generalizes Corollary 4 by allowing the subtype variable to appear anywhere in the context.

▶ **Lemma 9** (Stability of Subtyping, Generalized). *If* $\Psi \vdash A \leq B$ *and* $\Psi \vdash C$ *then* $\Psi \vdash [C/\tilde{a}]A \leq [C/\tilde{a}]B$.

This property ensures that any subtype variable can be replaced by a polytype C in two types A and B while preserving the subtyping relation between those two types.

The Subsumption Lemma. To prove the checking subsumption lemma, we first need to generalize the statement for inference and application inference judgments, as well as introduce a *context subtyping* relation, $\Psi \leq \Psi'$, to state the most general form.

▶ **Definition 10.** $\boxed{\Psi' \leq \Psi}$ *Context Subtyping*

$$\frac{}{\cdot \leq \cdot}\ \texttt{CS_Empty} \qquad \frac{\Psi' \leq \Psi}{\Psi', a \leq \Psi, a}\ \texttt{CS_TV} \qquad \frac{\Psi' \leq \Psi}{\Psi', \tilde{a} \leq \Psi, \tilde{a}}\ \texttt{CS_STV}$$

$$\frac{\Psi' \leq \Psi \quad \Psi \vdash A' \leq A}{\Psi', x : A' \leq \Psi, x : A}\ \texttt{CS_V}$$

Context Ψ subsumes context Ψ' if they bind the same variables in the same order, but the types of variables in Ψ' must be subtypes of those in Ψ. The generalized lemma is:

▶ **Lemma 11** (Subsumption). *Given* $\Psi' \leq \Psi$*:*
1. *If* $\Psi \vdash e \Leftarrow A$ *and* $\Psi \vdash A \leq A'$ *then* $\Psi' \vdash e \Leftarrow A'$*;*
2. *If* $\Psi \vdash e \Rightarrow B$ *then there exists* B' *s.t.* $\Psi \vdash B' \leq B$ *and* $\Psi' \vdash e \Rightarrow B'$*;*
3. *If* $\Psi \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow C$ *and* $\Psi \vdash A' \leq A$*, then* $\exists C'$ *s.t.* $\Psi \vdash C' \leq C$ *and* $\Psi' \vdash A' \bullet e \Rightarrow\!\!\!\Rightarrow C'$.

This lemma expresses that any derivation in a context Ψ has a corresponding derivation in any context Ψ' that it subsumes.

Type variables	a, b	Subtype variables	$\tilde{a}, \tilde{b}$	Existential variables	$\widehat{\alpha}, \widehat{\beta}$
Algorithmic types	A, B, C	$::=$	$\ldots \mid \widehat{\alpha}$		
Judgment chain	ω	$::=$	$A \leq B \mid e \Leftarrow A \mid e \Rightarrow_a \omega \mid A \bullet e \Rrightarrow_a \omega \mid A \circ B \Rrightarrow_a \omega$		
Algorithmic worklist	Γ	$::=$	$\cdot \mid \Gamma, a \mid \Gamma, \tilde{a} \mid \Gamma, \widehat{\alpha} \mid \Gamma, x : A \mid \Gamma \Vdash \omega$		
Declarative worklist	Ω	$::=$	$\cdot \mid \Omega, a \mid \Omega, \tilde{a} \mid \Omega, x : A \mid \Omega \Vdash \omega$		

Figure 4 Extended Syntax for the Algorithmic System (Extended from Figure 1).

Relating Subtype and Type Variables. The following lemma shows that we can substitute a subtype variable with a normal type variable, while preserving the subtyping relation.

▶ **Lemma 12.** *If* $\Psi[\tilde{a}] \vdash A \leq B$ *then* $\Psi[a] \vdash [a/\tilde{a}]A \leq [a/\tilde{a}]B$.

The reason is relatively straightforward. First, the substitution does not affect the $\tilde{a} \leq \tilde{a}$ sub-judgments. Second, substituting $\tilde{a}$ to a increases the range of implicit instantiation, which means that the monotypes picked in the old context are still well-formed under the new context. Note that the reverse statement does not hold. For example, $b \vdash \forall a.\ a \to a \leq b \to b$ holds with the predicative instantiation $a := b$, but $\tilde{b} \vdash \forall a.\ a \to a \leq \tilde{b} \to \tilde{b}$ does not; one cannot instantiate a with a non-monotype, or $\tilde{b}$ in this case.

4 Algorithmic System

This section introduces an algorithmic system that implements the syntax-directed specification of $F^e_{<:}$. The new algorithm is based on Zhao et al.'s [39] worklist algorithm but extended with explicit type applications and top and bottom types. In Section 5, we show that this algorithm is sound, complete and decidable with respect to the specification presented in Section 3. We also use color throughout this section to highlight differences to the original formulation by Zhao et al. [39].

4.1 Syntax and Well-Formedness

Figure 4 shows the syntax for the algorithmic version of $F^e_{<:}$. Similarly to the syntax-directed system, the well-formedness rules are unsurprising, ensuring well-scopedness for binders as well as the free variable constraint on polymorphic types. We refer to the extended version for well-formedness relations of algorithmic types, expressions, judgments, and worklists.

Existential Variables. The algorithmic system inherits the syntax of terms from the syntax-directed system and extends types with a new sort of variables – *existential variables*. Existential variables ($\widehat{\alpha}, \widehat{\beta}$) are introduced to help find unknown monotypes τ that appear in multiple rules of the syntax-directed system. In the algorithmic worklist, the position where existential variables are declared indicates the possible monotypes they can be solved to. Formally speaking, if $\widehat{\alpha}$ is introduced right after Γ, then $\widehat{\alpha}$ can only be solved to a monotype τ where $\Gamma \vdash \tau$. This behavior is derived from the well-formedness restriction of the rule $\leq\forall$L. An important remark is that subtype variables are not considered to be monotypes, therefore no existential variable can be solved to a subtype variable.

Judgment Chains. Judgment chains ω, or judgments for short, are the core components of our algorithmic type-checking. There are five kinds of judgments in our system. Four of them are inherited from [39]: subtyping ($A \leq B$), checking ($e \Leftarrow A$), inference ($e \Rightarrow_a \omega$) and application inference ($A \bullet e \Rrightarrow_a \omega$). Type application inference $A \circ B \Rrightarrow_a \omega$ is new, and it is used to help with the inference in type application expressions (e @B). This judgment plays a role similar to application inference for regular applications. In type application inference judgments, the first type A is the type inferred from the expression e. The judgment is then reduced differently depending on whether A is a polymorphic type $\forall a.\ A'$ or $\bot$.

Subtyping and checking are relatively simple, since their results are only success or failure. However, inference, application inference, and type application inference judgments return a type that is used in subsequent judgments. We use a continuation-passing-style encoding to accomplish this, following the approach by Zhao et al. [39]. For example, the judgment chain $e \Rightarrow_a (a \leq B)$ contains two judgments: first we infer the type of the expression e, and then check if the inferred type is a subtype of B. The *unknown* type of e is represented by a type variable a, which is used as a placeholder in the second judgment to denote the type of e.

Worklist Judgments. Our algorithmic context Γ, or *worklist*, combines traditional contexts and judgment(s) into a single sort. The worklist is an *ordered* collection of both variable bindings and judgments. The order captures the scope: only the objects that come after a variable's binding in the worklist can refer to it. For example, $[\cdot, a, x : a \Vdash x \Leftarrow a]$ is a valid worklist, but $[\cdot \Vdash \underline{x} \Leftarrow \underline{a}, x : \underline{a}, a]$ is not (the underlined symbols refer to out-of-scope variables). This property also affects how the algorithm behaves regarding solving existential variables. By solving an existential variable $\widehat{\alpha}$ with any monotype that does not escape the scope of $\widehat{\alpha}$ preserves well-formedness of the whole worklist.

Notation and Form of the Algorithmic Rules. The algorithmic subtyping and typing reduction rules, defined in Figures 5 and 6, have the form $\Gamma \longrightarrow \Gamma'$. Since the worklist is a stack of variable definitions and judgment chains, the algorithm pops the first element, processes according to the rules, and possibly pushes simplified judgments back. The syntax $\Gamma \longrightarrow^* \Gamma'$ denotes multiple reduction steps. A worklist Γ is accepted by the algorithm iff $\Gamma \longrightarrow^* \cdot$. In other words a program successfully type-checks if all the work has been processed. Any new variable introduced to the r.h.s of the worklist Γ' is fresh implicitly, similarly to how we treat them in the conditions of other rules. We also adopt the notation $\Gamma[\Gamma_M]$ from the DK type system to denote the worklist $\Gamma_L, \Gamma_M, \Gamma_R$, where $\Gamma[\bullet]$ is the worklist $\Gamma_L, \bullet, \Gamma_R$ with a hole ($\bullet$). Hole notations with the same name implicitly share the same structure Γ_L and Γ_R. A multi-hole notation splits the worklist into more parts. For example, $\Gamma[\widehat{\alpha}][\widehat{\beta}]$ means $\Gamma_1, \widehat{\alpha}, \Gamma_2, \widehat{\beta}, \Gamma_3$.

4.2 Garbage Collection and Algorithmic Subtyping Rules

Figure 5 defines algorithmic rules on variables (garbage collection) and subtyping. Rules 1-4 pop variable declarations that are essentially garbage. Thanks to the nature of ordered context, those variables are no longer referred to by the remaining judgments, therefore removing them does not break the well-formedness of the worklist.

Subtyping rules. We can discern 3 groups of rules for algorithmic subtyping. The first group consists of rules 5-12, where all the rules are similar to their syntax-directed system counterparts. The most interesting one is rule 12, which reflects the changes in our syntax-directed system. A subtype variable $\tilde{a}$ is used to replace the bound variable in the polymorphic types $\forall a.\ A$ and $\forall a.\ B$ for further reduction. Rule 11 differs from rule $\leq\forall$L by introducing an existential variable $\widehat{\alpha}$ instead of guessing the monotype τ instantiation.

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ reduces to Γ'.

$$\Gamma, a \longrightarrow_1 \Gamma \qquad \Gamma, \widehat{\alpha} \longrightarrow_2 \Gamma \qquad \Gamma, \tilde{a} \longrightarrow_3 \Gamma \qquad \Gamma, x : A \longrightarrow_4 \Gamma$$

$$\begin{aligned}
\Gamma \Vdash 1 \leq 1 \longrightarrow_5 \Gamma \qquad \Gamma \Vdash a \leq a &\longrightarrow_6 \Gamma \qquad \Gamma \Vdash \tilde{a} \leq \tilde{a} \longrightarrow_7 \Gamma \\
\Gamma \Vdash A \leq \top &\longrightarrow_8 \Gamma \qquad \Gamma \Vdash \bot \leq A \longrightarrow_9 \Gamma \\
\Gamma \Vdash A_1 \to A_2 \leq B_1 \to B_2 &\longrightarrow_{10} \Gamma \Vdash A_2 \leq B_2 \Vdash B_1 \leq A_1 \\
\Gamma \Vdash \forall a.\ A \leq B &\longrightarrow_{11} \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \leq B \\
& \quad \text{when } B \neq \forall a.\ B' \text{ and } B \neq \top \\
\Gamma \Vdash \forall a.\ A \leq \forall a.\ B &\longrightarrow_{12} \Gamma, \tilde{a} \Vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash \widehat{\alpha} \leq \widehat{\alpha} &\longrightarrow_{13} \Gamma \\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B &\longrightarrow_{14} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B) \\
& \quad \text{when } \widehat{\alpha} \notin FV(A) \cup FV(B) \\
\Gamma[\widehat{\alpha}] \Vdash A \to B \leq \widehat{\alpha} &\longrightarrow_{15} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash A \to B \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2) \\
& \quad \text{when } \widehat{\alpha} \notin FV(A) \cup FV(B)
\end{aligned}$$

$$\begin{aligned}
\Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\alpha} \leq \widehat{\beta} &\longrightarrow_{16} [\widehat{\alpha}/\widehat{\beta}](\Gamma[\widehat{\alpha}][]) \qquad & \Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\beta} \leq \widehat{\alpha} &\longrightarrow_{17} [\widehat{\alpha}/\widehat{\beta}](\Gamma[\widehat{\alpha}][]) \\
\Gamma[a][\widehat{\beta}] \Vdash a \leq \widehat{\beta} &\longrightarrow_{18} [a/\widehat{\beta}](\Gamma[a][]) & \Gamma[a][\widehat{\beta}] \Vdash \widehat{\beta} \leq a &\longrightarrow_{19} [a/\widehat{\beta}](\Gamma[a][]) \\
\Gamma[\widehat{\beta}] \Vdash 1 \leq \widehat{\beta} &\longrightarrow_{20} [1/\widehat{\beta}](\Gamma[]) & \Gamma[\widehat{\beta}] \Vdash \widehat{\beta} \leq 1 &\longrightarrow_{21} [1/\widehat{\beta}](\Gamma[])
\end{aligned}$$

Figure 5 Algorithmic Variable and Subtyping Rules.

The second group is about solving existential variables (rule 13) and existential variable decomposition (rules 14 and 15). Rule 13 is one of the base cases involving existential variables. Rules 14 and 15 are algorithmic versions of Rule $\leq\!\to$; they both partially instantiate $\widehat{\alpha}$ to function types. The domain $\widehat{\alpha}_1$ and range $\widehat{\alpha}_2$ of the new function type are not determined immediately: they are fresh existential variables with the same scope as $\widehat{\alpha}$. The *occurs-check* condition prevents divergence as usual. For example, without it $\widehat{\alpha} \leq 1 \to \widehat{\alpha}$ would diverge.

The final group consists of rules 16-21, where each rule solves an existential variable against a basic type. Each rule removes an existential variable and substitutes it with its solution in the remaining worklist, which preserves well-formedness in the meantime. For example, Rule 16 solves variable $\widehat{\alpha}$ with $\widehat{\beta}$ only if $\widehat{\beta}$ occurs after $\widehat{\alpha}$. It is worth noting that none of these rules solves $\widehat{\alpha}$ to a subtype variable $\tilde{b}$. As we have discussed, $\top$, $\bot$ and subtype variables are not monotypes, therefore existential variables do not unify with them.

4.3 Algorithmic Typing Rules

Figure 6 shows the algorithmic rules for typing.

Checking Judgments. Rules 22-26 deal with checking judgments. Rule 22 is `DSub` written in a continuation-passing-style. The side conditions $e \neq \lambda x.\ e'$ and $B \neq \top$ prevent overlap with all other rules. Rules 23, 24 and 26 adapt their counterparts in the syntax-directed system, where rules 23 and 26 correspond to the new/changed rules introduced in the syntax-directed system compared to DK's work. Rule 25 is a special case of `D→I`, dealing with the case when the input type is an existential variable, representing a monotype *function* as in the syntax-directed system. The same instantiation technique as in rules 14 and 15 applies.

$\boxed{\Gamma \longrightarrow \Gamma'}$ (cont.) Γ reduces to Γ'.

$$
\begin{aligned}
\Gamma \Vdash e \Leftarrow B &\longrightarrow_{22} \Gamma \Vdash e \Rightarrow_a a \leq B \\
&\qquad \text{when } e \neq \lambda x.\ e' \text{ and } B \neq \forall a.\ B' \text{ and } B \neq \top \\
\Gamma \Vdash e \Leftarrow \forall a.\ A &\longrightarrow_{23} \Gamma, a \Vdash e \Leftarrow A \\
\Gamma \Vdash \lambda x.\ e \Leftarrow A \to B &\longrightarrow_{24} \Gamma, x : A \Vdash e \Leftarrow B \\
\Gamma[\widehat{\alpha}] \Vdash \lambda x.\ e \Leftarrow \widehat{\alpha} &\longrightarrow_{25} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2], x : \widehat{\alpha}_1 \Vdash e \Leftarrow \widehat{\alpha}_2) \\
\Gamma \Vdash e \Leftarrow \top &\longrightarrow_{26} \Gamma \\[1em]
\Gamma \Vdash x \Rightarrow_a \omega &\longrightarrow_{27} \Gamma \Vdash [A/a]\omega \quad \text{when } (x : A) \in \Gamma \\
\Gamma \Vdash (e : A) \Rightarrow_a \omega &\longrightarrow_{28} \Gamma \Vdash ([A/a]\omega) \Vdash e \Leftarrow A \\
\Gamma \Vdash (\Lambda a.\ e : A) \Rightarrow_b \omega &\longrightarrow_{29} \Gamma \Vdash ([\forall a.\ A/b]\omega), a \Vdash e \Leftarrow A \\
\Gamma \Vdash () \Rightarrow_a \omega &\longrightarrow_{30} \Gamma \Vdash [1/a]\omega \\
\Gamma \Vdash \lambda x.\ e \Rightarrow_a \omega &\longrightarrow_{31} \Gamma, \widehat{\alpha}, \widehat{\beta} \Vdash ([\widehat{\alpha} \to \widehat{\beta}/a]\omega), x : \widehat{\alpha} \Vdash e \Leftarrow \widehat{\beta} \\
\Gamma \Vdash e_1\ e_2 \Rightarrow_a \omega &\longrightarrow_{32} \Gamma \Vdash e_1 \Rightarrow_b (b \bullet e_2 \Rightarrow\!\!\!\Rightarrow_a \omega) \\
\Gamma \Vdash e\ @C \Rightarrow_a \omega &\longrightarrow_{33} \Gamma \Vdash e \Rightarrow_b (b \circ C \Rightarrow\!\!\!\Rightarrow_a \omega) \\[1em]
\Gamma \Vdash \forall b.\ B \circ C \Rightarrow\!\!\!\Rightarrow_a \omega &\longrightarrow_{34} \Gamma \Vdash [([C/b]B)/a]\omega \\
\Gamma \Vdash \bot \circ C \Rightarrow\!\!\!\Rightarrow_a \omega &\longrightarrow_{35} \Gamma \Vdash [\bot/a]\omega \\[1em]
\Gamma \Vdash A \to C \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega &\longrightarrow_{36} \Gamma \Vdash ([C/a]\omega) \Vdash e \Leftarrow A \\
\Gamma \Vdash \forall a.\ A \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega &\longrightarrow_{37} \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega \\
\Gamma \Vdash \bot \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega &\longrightarrow_{38} \Gamma \Vdash [\bot/a]\omega \\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega &\longrightarrow_{39} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega)
\end{aligned}
$$

Figure 6 Algorithmic Typing Rules.

Inference judgments. Inference judgments accept an expression and return a type. Rules 27-33 deal with type inference judgments. The algorithm uses a continuation-passing-style encoding, where the output type is passed to the next judgment. When an inference judgment succeeds with type A, the algorithm continues to work on the inner-chain ω by substituting a by A in ω. Rule 27 and 30 are base cases (variable and unit), where the inferred type is passed to its child judgment chain. Rules 28 and 29 infer an annotated expression by changing into checking mode, therefore another judgment chain is created. Rule 29 deals with scoped type variables; the type variable a is in scope in e, and corresponds to the rule `DTA∀`. Rule 31 infers the type of a lambda expression by introducing $\widehat{\alpha}, \widehat{\beta}$ as the input and output types of the function, respectively. Rule 32 infers the type of an application by firstly inferring the type of the function e_1. Then the remaining work is delegated to an application inference judgment, which passes a, representing the return type of the application, to the remainder of the judgment chain ω. Rule 33 is new: it first infers the type of e, then calls the type application inference judgment to compute the return type.

Type Application and Application Inference Judgments. Rules 34 and 35 deal with the new type application inference judgments. Rule 34 accepts a polymorphic input $\forall b.\ B$ and produces its instantiation $[C/b]B$. Rule 35 returns $\bot$ as it can be used as any type. For example, if we choose to treat $\bot$ as the polymorphic type $\forall b.\ \bot$, the result after type application is $\bot$ according to rule 34. Finally, Rules 36-39 deal with application inference judgments. Rules 36, 37 and 38 behave like rules $\mathtt{D\forall App}$, $\mathtt{D{\rightarrow}App}$ and $\mathtt{D\bot App}$, respectively. Rule 39 instantiates $\widehat{\alpha}$ to the function type $\widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2$, just like Rules 14, 15 and 25.

5 Algorithmic Metatheory

This section presents the metatheory of the algorithmic system in Section 4. We show three main results: *soundness*, *completeness* and *decidability*. Our proofs employ similar techniques to the ones by Zhao et al. [39], so we only highlight the main results and differences.

5.1 Declarative Worklist and Transfer

To aid in formalizing the correspondence between the declarative and algorithmic systems, we use a declarative worklist Ω, defined in Figure 4. A declarative worklist Ω has the same structure as an algorithmic worklist Γ, but does not contain any existential variables $\widehat{\alpha}$.

Worklist Instantiation. We instantiate an algorithmic worklist Γ to the declarative worklist Ω by instantiating all existential variables $\widehat{\alpha}$ in Γ with well-scoped monotypes τ.

▶ **Definition 13.** $\boxed{\Gamma \rightsquigarrow \Omega}$ Γ *instantiates to* Ω.

$$\frac{}{\Omega \rightsquigarrow \Omega}\ {\rightsquigarrow}\Omega \qquad \frac{\Omega \vdash \tau \qquad \Omega, [\tau/\widehat{\alpha}]\Gamma \rightsquigarrow \Omega}{\Omega, \widehat{\alpha}, \Gamma \rightsquigarrow \Omega}\ {\rightsquigarrow}\widehat{\alpha}$$

Rule ${\rightsquigarrow}\widehat{\alpha}$ replaces the first (left-most) existential variable with a well-scoped monotype and repeats the process on the resulting worklist until no existential variable remains and thus the algorithmic worklist has become a declarative one. In order to maintain well-scopedness, the substitution is applied to all the judgments and term variable bindings in the scope of $\widehat{\alpha}$.

Declarative Worklist Reduction. A relation $\Omega \longrightarrow \Omega'$ is defined to reduce all judgments in the declarative worklists with declarative typing rules. This relation checks that every judgment entry in the worklist holds using a corresponding conventional declarative judgment. The typing contexts of declarative judgments are recovered using an auxiliary erasure function $\|\Omega\|$. The erasure function simply drops all judgment entries from the worklist, keeping only variable and type variable declarations. Both definitions are available in the extended version.

5.2 Soundness

Our algorithm is sound with respect to the declarative system. For any worklist Γ that reduces successfully, there is a valid instantiation Ω that transfers all judgments to the declarative system.

▶ **Theorem 14** (Soundness). *If* wf Γ *and* $\Gamma \longrightarrow^* \cdot$, *then* $\exists \Omega$ *s.t.* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$.

The proof proceeds by induction on the derivation of $\Gamma \longrightarrow^* \cdot$. Most of the proof follows Zhao et al. [39]. Algorithmic type application rules are the most interesting change, because they have a different shape compared to the declarative rules. With the help of declarative worklist reduction, we can reduce the additional form of type application syntax and therefore indirectly build a relationship with the declarative system.

5.3 Completeness

Any derivation in the declarative system has an algorithmic counterpart:

▶ **Theorem 15** (Completeness). *If* wf Γ *and* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$*, then* $\Gamma \longrightarrow^* \cdot$*.*

We prove completeness by induction on the derivation of $\Omega \longrightarrow^* \cdot$ with a similar technique to the one used by Zhao et al. [39]. New rules, including the ones involve subtype variables and type applications, do not increase the difficulty of our proof significantly. It is worth noting that our system forbids the $\top$ and $\bot$ types to be instantiated by monotypes. If we did not pose such restriction, then the following lemma would not hold anymore:

▶ **Lemma 16** (Prune Transfer for Instantiation). *If* $(\Gamma \Vdash \widehat{\alpha} \le A \to B) \rightsquigarrow (\Omega \Vdash C \le A_1 \to B_1)$ *and* $\|\Omega\| \vdash C \le A_1 \to B_1$*, then* $\widehat{\alpha} \notin FV(A) \cup FV(B)$*.*

For example, allowing instantiations like $\widehat{\alpha} := \top$ would make the algorithmic judgment $\widehat{\alpha} \to \widehat{\alpha} \le \widehat{\alpha}$ derivable. This lemma is essential to follow the original proof to prove completeness for the occurs-check condition in rules 14 and 15.

5.4 Decidability

Finally, we show that our algorithm is decidable:

▶ **Theorem 17** (Decidability). *Given* wf Γ*, it is decidable whether* $\Gamma \longrightarrow^* \cdot$ *or not.*

Our decidability proof is based on a lexicographic group of induction measures:

$$\langle |\Gamma|_e, |\Gamma|_{\Leftrightarrow}, |\Gamma|_{\top\bot}, |\Gamma|_{\forall}, |\Gamma|_{\widehat{\alpha}}, |\Gamma|_{\to} + |\Gamma| \rangle$$

on the worklist Γ. Compared with the measures used by Zhao et al. [39], we introduce a new measure $|\cdot|_{\top\bot}$, which counts the total number of $\top$ and $\bot$ occurrences. This is required because judgments like $\widehat{\alpha} \le \top$ now do not solve $\widehat{\alpha}$, which breaks the original proof technique. This type of judgment now reduces the new measure by at least one. The rest of the proof follows the approach closely. The extended version has detailed explanations of the measures and proofs. Combining all three main results (soundness, completeness and decidability), we conclude that the declarative system is decidable by means of our algorithm.

▶ **Corollary 18** (Decidability of Declarative Typing). *Given* wf Ω*, it is decidable whether* $\Omega \longrightarrow^* \cdot$ *or not.*

6 Discussion

Inferring Top and Bottom Types. $F^e_{<:}$ does not treat the $\top$ and $\bot$ types as monotypes, therefore these types cannot be implicitly instantiated by the type inference algorithm. However, in certain programming languages, especially OOP languages with downcasts, the $\top$ type can be useful in certain cases and implicit instantiation would be convenient to have. For example, the following Java program

```
var ns = List.of(1, 2, "3");
```

should instantiate the generic variable `A` of the `List<A>` class to `Object` (note that `Object` plays a similar role to $\top$ in Java) to type-check the program. Thus, the inferred type for `ns` is `List<Object>`. In this program, because downcasts are possible in Java, it is plausible that the programmer intended to have a heterogeneous list of values, that could later be accessed by doing some type analysis for the elements and downcasting from `Object` to `Integer` or `String`. We consider such use cases to be a practical example where instantiation with the top type would be useful in languages like Java. In contrast, in $F^e_{<:}$, we would need to explicitly instantiate the type argument. Nonetheless, in a language without downcasts (such as $F^e_{<:}$), the declaration of `ns` above would very likely be a programmer error, since there would not be much that could be done with a value of type `List<Object>`. As we have argued in Section 2.4, there is a tension between inferring types with top and bottom types and hiding programmer errors: sometimes type errors that would be caught in many type systems, are instead type-checked by inferring some types with top and bottom types. Our design decision in $F^e_{<:}$ is not to infer top and bottom types, which avoids hiding such errors as well as avoiding the technical complexities that arise from inferring such types.

It is possible to have alternative designs for $F^e_{<:}$ that infer top and bottom types as well. For instance, if we would be aiming at covering common cases that arise in practice in languages like Java, such as the inference of the type of `ns` above, we could extend our syntax-directed system and algorithmic system with the rules:

$$\frac{\Psi \vdash [\top/a]A \leq B \quad B \neq \top}{\Psi \vdash \forall a.\ A \leq B}\ \texttt{DInst}\top$$

$$\Gamma \Vdash \forall a.\ A \leq B \longrightarrow \Gamma \Vdash [\top/a]A \leq B \quad \text{when } B \neq \top$$

The two rules above support simple forms of instantiation where the type variable is directly instantiated with the $\top$ type (a similar approach could be used for $\bot$ types). Note that these rules *overlap* with the current predicative instantiation rules, and thus introduce nondeterminism. Implementing the algorithmic rule directly would require some backtracking. From the theoretical point of view the rules are quite ad-hoc, since they cover only very specific cases of instantiation with top types. A more theoretically appealing approach would be to borrow ideas from approaches such as MLSub [6], which can infer types with top and bottom. However, this would be much more technically challenging. Another direction would be to complement the global type inference approach of $F^e_{<:}$ with some more local approach to attempt to infer top and bottom types. We will discuss this approach more next.

Local Impredicative Inference. Implicit impredicative instantiation is an advanced feature in modern type systems, and it is also supported by the local type inference approach [27]. Unlike top types, which can have some practical use cases in languages like Java, no existing mainstream OOP languages support higher-ranked systems with first-class polymorphic functions/values. Thus, there is no need for impredicative instantiations in those languages today. Nonetheless, future languages may support such feature and it is worthwhile considering impredicative type inference. Local type inference algorithms are designed to support impredicative instantiations through information in the neighbor nodes of the syntax tree. The recent work on Quick Look by [32] instantiates polymorphic types through a similar local approach and falls back to a global Hindley-Milner-style unification afterwards. Currently, our system only employs global unification and ignores any local information. We believe that a promising direction would be to follow the Quick Look approach, preserve the core global

inference system of $F^e_{<:}$, and try to employ a more local approach to infer impredicative types as well as top and bottom types before introducing unification variables. The main challenge in this direction is that Serrano et al.'s [32] approach relies on invariant subtyping for function types. In contrast, we have to deal with contravariance for input types and covariance for output types.

7 Related Work

This section discusses related work, focusing on the most closely related research on higher-ranked type inference and local type inference.

Hindley-Milner. The Hindley-Milner (HM) type system [5, 19, 16] was a landmark achievement in type inference. The constraint-based presentation by Pottier and Rémy [29] for HM and ML type inference has similarities with the worklist approach and it also keeps precise scoping of variables. In HM the order of universally quantified variables is irrelevant and no annotations are required. In contrast, in our work, the order of universally quantified variables matters, and annotations are necessary for polymorphic functions. Thus, we do not support Hindley-Milner style generalization. Nevertheless if we assume annotations of polymorphic expressions, the order-relevance of universally quantified variables is not problematic. Because of its support for visible type applications [13], GHC Haskell already distinguishes between *specified* and *generalized* type quantification. Specified type quantification refers to polymorphic expressions that have explicit type annotations. Like $F^e_{<:}$, in GHC type variables in specified quantification are order relevant to be compatible with explicit type applications. In contrast to Hindley-Milner, $F^e_{<:}$ supports higher-ranked polymorphism, explicit impredicative type applications and top and bottom types.

Higher-Ranked Polymorphic Type Inference. There has been much work extending HM while preserving all of its expressive power. In particular, there are several extensions of HM to System F, which support *higher-ranked polymorphism*. Since full type inference for System F is undecidable [38], such extensions need some type annotations or restrictions to remain decidable. The work on type inference for higher-ranked polymorphism (HRP) can be divided into two main lines: predicative and impredicative type systems. In predicative type systems, only monotypes can be inferred. An advantage of predicative type systems is that the predicative polymorphic subtyping relation is decidable [20], which facilitates the design of such type systems and type inference algorithms. There are several predicative HRP type systems [25, 8, 9, 20, 39]. The work in this paper is based on DK's [8] declarative type system and the algorithmic formulation by Zhao et al. [39]. However, we support explicit impredicative type applications and top and bottom types. Such features create various challenges and, to address some of those challenges, we introduce a novel stable polymorphic subtyping relation. In contrast, DK adopt the polymorphic subtyping relation by Odersky and Läufer [20]. In essence, with stable subtyping, the order of type variables becomes relevant in universal quantification. As a consequence, some forms of subtyping that are accepted by Odersky and Läufer's relation are rejected in our type system. Nonetheless, with those restrictions we retain important properties, such as checking subsumption and stability of type substitutions, in the presence of new features that are not supported by DK.

Impredicative System F allows instantiation with polymorphic types. Unfortunately, a subtyping relation with impredicative implicit instantiation is undecidable [3, 35]. Work on partial impredicative type inference algorithms [17, 18, 36, 33, 32, 14] navigate a variety of

design tradeoffs for a decidable algorithm. Ideas from *Guarded Impredicative Polymorphism* [33] and the *Quick Look* approach [32], are being adopted in GHC 9 for enabling impredicative instantiation. They make use of local information in n-ary applications to infer polymorphic instantiations with a relatively simple specification and unification algorithm. Although not all impredicative instantiations can be handled well, these approaches are useful in practice. In contrast to this line of work, we do not attempt to infer impredicative types. Instead, all impredicative instantiations must be explicit. While explicit instantiation is less convenient, an advantage is flexibility. Approaches that only allow implicit impredicative instantiation may reject some instantiations that would be possible with explicit instantiation.

Stability. Besides the motivation of supporting a form of impredicative polymorphism, another motivation for the changes in type inference in GHC 9 has been to simplify the algorithms and address various issues surrounding subsumption. While we are not aware that the issues that we have described in Section 2 have been previously identified, there have been several discussions documenting other issues related to subsumption in GHC 8 [26]. Recently, motivated to understand what would be the best design for instantiation in GHC, Bottu and Eisenberg [1] have compared four different approaches to instantiation. They have identified stability properties as an important factor for language designers to take into consideration when designing languages with implicit instantiation. Stability also plays an important role in the Cochis calculus [31], where it ensures that the behavior of resolution (which is a mechanism employed by type classes [37] or Scala implicits [22]) is preserved after instantiation. Stable subtyping in $F^e_{<:}$ provides a high-level specification of polymorphic subtyping, which essentially embeds a stability property into the subtyping relation.

Type Inference with Explicit Type Applications. The work on *visible type application* (VTA) [13] adds a predicative form of explicit type application to HM and HRP type systems. This approach has been adopted in GHC 8. As discussed in detail in Section 2, a property that is not enforced in VTA is checking subsumption. We believe that checking subsumption is an important property, as it ensures that a program can always be annotated with a supertype and it can prevent situations where simply inlinings of function definitions can make a well-typed program ill-typed. The Quick Look approach [32] supports impredicative visible type application. An important difference is that in Quick Look subtyping of functions is invariant, whereas in the original VTA approach the standard subtyping rule is used. The invariant subtyping rule prevents the counter-examples to checking subsumption that we found in GHC 8, and described in Section 2. Our work shows a different way to prevent such examples, by employing stable subtyping with a standard subtyping rule for functions. We believe that there are merits in both approaches. The restrictions adopted by Quick Look do not affect backward compatibility with the HM type system. In contrast, elementary type inference does not aim at backwards compatibility with the HM type system. Instead, we are interested in backward compatibility with extensions of System F with subtyping (such as $F_{<:}$ [2]). Quick Look would not preserve backward compatibility to such type systems, which employ a standard subtyping rule for function types.

Local Type Inference. While technically speaking we are closest to predicative HRP, we are closer in spirit and in goals to local type inference [27, 21]. Like our work, local type inference does not aim to subsume the HM type system. Local type inference sacrifices some of the expressive power of type inference, in exchange for the ability to smoothly deal with features such as top and bottom types and impredicative types. Pierce and Turner

considered a language similar to the language that we consider in our work (with top and bottom types, but no bounded quantification). Like our approach, both implicit and explicit type applications are supported. Technically speaking, our approach is still a global inference approach, and thus it is quite different from local type inference. In local type inference, missing annotations are recovered using only information from adjacent nodes in the syntax tree, and there are no long-distance constraints such as unification variables. We believe that an advantage of $F^e_{<:}$ is that it has simple and clear syntax-directed specifications, whereas the specification of local type inference is more involved, and it is not obvious to programmers when instantiation works or not. Furthermore, $F^e_{<:}$ allows the inference of lambda expressions without any contextual type information, as long as the inferred type is a monotype.

Type Inference with Subtyping. Another line of work is extensions of HM with subtyping. Type systems in presence of subtyping encounter constraints that are not simply equalities as in HM. Therefore constraint solvers used in HM, where unifications are based on equality, cannot be easily extended to support subtyping. Instead, constraints are usually collected as subtyping relations and may delay resolution as the constraints accumulate. Some systems that are based on *constraint types* [12, 11], i.e. types expressed together with a set of constraints $\tau \mid \{\overline{\tau_1 \leq \tau_2}\}$. Unfortunately, such constraints can be quite large and hard to interpret by programmers. Pottier [30] proposed three methods to simplify constraints, aiming at improving the efficiency of type inference algorithms and improving the readability of the resulting types. Inspired by the simplification strategies of Pottier, MLsub [6] suggests that the data flow on the constraint graph can be reflected directly on types in a richer type system. Simple-sub [23] further simplifies the algorithm of MLsub and is implemented in 500 lines of code. While being equivalent to MLsub, it is a more efficient variant. In our work, we avoid subtyping constraints and do not infer types with top and bottom types. If instantiations with such types are needed, then an explicit type application must be used. On the other hand, we support higher-ranked polymorphism, and explicit type applications, which (as far as we know) are not supported by any extensions of HM with subtyping.

8 Conclusion

In this paper, we proposed elementary type inference: a partial form of type inference that can be used in languages with subtyping that combine implicit instantiation with explicit type applications. As type systems become more powerful, the inference problem becomes harder, quickly leading to various undecidable problems. It is clear that some form of type-inference is needed in most languages to make their use practical. However, it is not necessarily true that being able to infer more types is always better, especially if there is the possibility to resort to explicit instantiation. Attempting to infer more types may have the side-effect of hiding programmer errors, as very general types can be inferred in the presence of advanced type system features. Moreover, *predicatibility* of what can be inferred and what cannot is also an important factor for users of the programming language. Elementary type inference strikes a compromise. It chooses to infer only monotypes, which are always inferrable, and makes it easy to understand when the instantiation succeeds or fails. For polytypes (which include top and bottom), explicit type applications must be used, but no expressive power is sacrificed. More work is needed to understand what is the right balance between inference, predicatibility and usability of languages in the future.

References

1. Gert-Jan Bottu and Richard A. Eisenberg. Seeking stability by being lazy and shallow: Lazy and shallow instantiation is user friendly. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, pages 85–97, New York, NY, USA, 2021. Association for Computing Machinery.
2. L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1):4–56, 1994.
3. Jacek Chrząszcz. Polymorphic subtyping without distributivity. In Luboš Brim, Jozef Gruska, and Jiří Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, pages 346–355, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
4. Maurizio Cimadamore. Jep 101: Generalized target-type inference, 2015. URL: `http://openjdk.java.net/jeps/101`.
5. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, 1982.
6. Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 60–72, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3009837.3009882`.
7. Jana Dunfield. Elaborating intersection and union types. *J. Funct. Program.*, 24(2-3):133–165, 2014. `doi:10.1017/S0956796813000270`.
8. Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, 2013.
9. Jana Dunfield and Neelakantan R. Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *PACMPL*, (POPL), January 2019. `arXiv:1601.05106`.
10. Jana Dunfield and Neelakantan R. Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.
11. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1995, Austin, Texas, USA, October 15-19, 1995*, OOPSLA '95, pages 169–184, New York, NY, USA, 1995. Association for Computing Machinery. `doi:10.1145/217838.217858`.
12. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1:132–153, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. `doi:10.1016/S1571-0661(04)80008-2`.
13. Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. Visible type application. In Peter Thiemann, editor, *Programming Languages and Systems*, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
14. Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. Freezeml: Complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 423–437. Association for Computing Machinery, 2020.
15. Andrew Gacek. The Abella interactive theorem prover (system description). In *Proceedings of IJCAR 2008*, Lecture Notes in Artificial Intelligence, 2008.
16. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.
17. Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, 2003.

18 Daan Leijen. HMF: Simple type inference for first-class polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, 2008.

19 Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

20 Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, 1996.

21 Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 41–53, New York, NY, USA, 2001. Association for Computing Machinery.

22 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, 2010.

23 Lionel Parreaux. The simple essence of algebraic subtyping: Principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. `doi:10.1145/3409006`.

24 Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Draft, 2004. URL: `http://research.microsoft.com/en-us/um/people/simonpj/papers/scoped-tyvars/`.

25 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(1):1–82, 2007.

26 Peyton Jones, Simon. Simplify Subsumption. *GHC Proposals*, 2020. URL: `https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0287-simplify-subsumption.rst`.

27 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

28 Hubert Plociniczak. *Decrypting Local Type Inference*. PhD thesis, EPFL, 2016.

29 François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference, pages 387–489. The MIT Press, 2005.

30 François Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, INRIA, 1998.

31 Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. COCHIS: Stable and coherent implicits. *Journal of Functional Programming*, 29:e3, 2019.

32 Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. A quick look at impredicativity. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

33 Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, 2018.

34 Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 203–216, New York, NY, USA, 2002. Association for Computing Machinery. `doi:10.1145/503272.503292`.

35 Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, 1996.

36 Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH: First-class polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, 2008.

37 P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. Association for Computing Machinery.

38 Joe B Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.

39 Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs

Madhurima Chakraborty ✉
University of California, Riverside, CA, USA

Renzo Olivares ✉
University of California, Riverside, CA, USA

Manu Sridharan ✉
University of California, Riverside, CA, USA

Behnaz Hassanshahi ✉
Oracle Labs, Brisbane, Australia

Abstract

Building sound and precise static call graphs for real-world JavaScript applications poses an enormous challenge, due to many hard-to-analyze language features. Further, the relative importance of these features may vary depending on the call graph algorithm being used and the class of applications being analyzed. In this paper, we present a technique to *automatically* quantify the relative importance of different root causes of call graph unsoundness for a set of target applications. The technique works by identifying the dynamic function data flows relevant to each call edge missed by the static analysis, correctly handling cases with multiple root causes and inter-dependent calls. We apply our approach to perform a detailed study of the recall of a state-of-the-art call graph construction technique on a set of framework-based web applications. The study yielded a number of useful insights. We found that while dynamic property accesses were the most common root cause of missed edges across the benchmarks, other root causes varied in importance depending on the benchmark, potentially useful information for an analysis designer. Further, with our approach, we could quickly identify and fix a recall issue in the call graph builder we studied, and also quickly assess whether a recent analysis technique for Node.js-based applications would be helpful for browser-based code. All of our code and data is publicly available, and many components of our technique can be re-used to facilitate future studies.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases JavaScript, call graph construction, static program analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.3

Related Version *Full Version*: https://arxiv.org/abs/2205.06780

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.7

Funding This research was supported in part by a gift from Oracle Labs and by the National Science Foundation under grant CCF-2007024. This research was partially sponsored by the OUSD(R&E)/RT&L and was accomplished under Cooperative Agreement Number W911NF-20-2-0267. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL and OUSD(R&E)/RT&L or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

© Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Effective call graph construction is critically important for JavaScript static analysis, as JavaScript analysis tools often need to reason about behaviors that span function boundaries (e.g., security vulnerabilities [26, 27] or correctness of library updates [40]). Unfortunately, call graph construction for real-world JavaScript programs poses significant challenges, particularly for client-side code in web applications. Modern web applications are increasingly built using sophisticated frameworks like React [4] and AngularJS [6].[1] Sophisticated recent JavaScript static analysis frameworks [32, 33, 36, 52] often focus on sound and precise handling of complex JavaScript constructs. While these systems have advanced significantly, they cannot yet scale to handle modern web frameworks. There are also a growing number of unsound but pragmatic call graph analyses designed primarily to give useful results for real-world code bases [8, 25, 40, 44]. While these techniques have been shown effective in certain domains, their unsoundness can lead to missing many edges when analyzing framework-based applications [27], i.e., the analyses can have low *recall*. For bug-finding and security analyses, these missing edges are of key concern as they can lead to false negatives like missed vulnerabilities.

To guide development of better call graph builders, it would be highly useful to know which language constructs are contributing most to reducing recall for a set of benchmarks of interest. JavaScript has many different constructs that are typically ignored or only partially handled by pragmatic static analyses, due to their dynamic nature [49]. Further, there are complex tradeoffs involved in adding support for these constructs, as a more complete handling may lead to scalability and precision problems. Analysis designers aiming to improve results for a set of benchmarks would be helped by quantitative guidance on the relative importance of different unhandled language features.

This paper presents a novel technique for *automatic root cause quantification* for missing edges in JavaScript call graphs. Figure 1 gives an overview of our technique. Given a program, a static call graph builder enhanced to also export static flow graphs (see Section 2.2), and a harness for exercising the program, our technique automatically finds *missing flows*, data flows of function values that occur at runtime but are not modeled by the static analysis. Our technique associates a set of missing flows with each missed call graph edge, thereby indicating which data flows must be handled by the static analysis to discover the missed edge. The technique correctly accounts for *inter-dependent calls*, where a call graph edge is missing due to the absence of other call graph edges.

We further observe that given a missing flow, one can often automatically determine a *root cause label* for the flow, indicating which unhandled language construct(s) were responsible for the flow being missed. Such labeling can be performed at different levels of granularity, depending on what level of detail is desired by the analysis designer. Given logic to map missing flows to root cause labels, our technique automatically quantifies the prevalence of each root cause for the desired benchmarks.

We have implemented our techniques, and we used them to study the recall of two variants of the approximate call graphs (ACG) algorithm of Feldthaus et al. [25], as implemented in the WALA framework [58], on a suite of modern web applications. We found the root cause quantification to provide useful insights, in particular:

- To our surprise, a large initial cause of low recall was the lack of models in WALA for a variety of built-in library functions. By adding models, we were able to increase recall by up to 5 percentage points.

[1] A recent Stack Overflow developer survey shows popularity of these frameworks is growing, with total usage surpassing older libraries like jQuery [56].

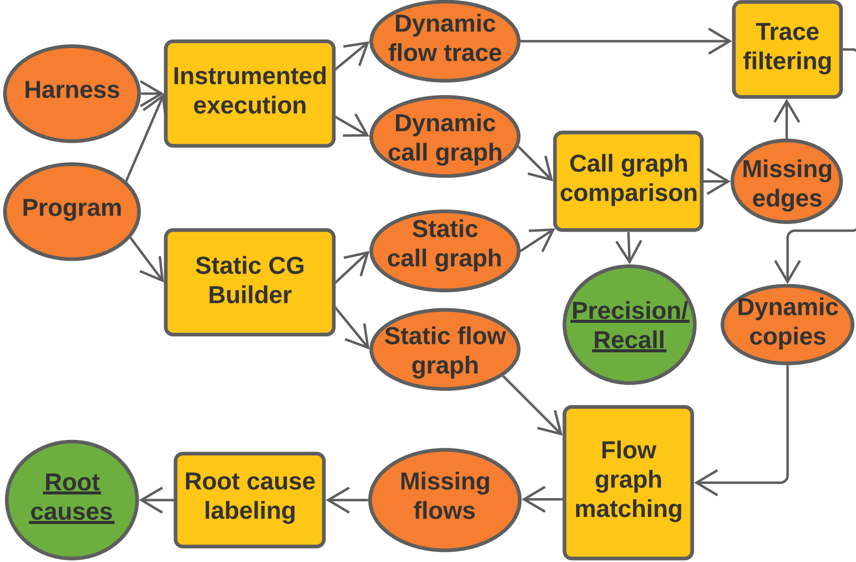


Figure 1 Overview of our methodology.

- After fixing the native models, dynamic property accesses were the largest root cause of low recall, at 70%. The second-largest root cause varied significantly across the benchmarks.
- We applied a finer-grained root cause labeling for dynamic property accesses, and found that their property names are computed in a wide variety of ways, with no single dominant pattern. We studied the potential of a recently-described recall-improving technique for dynamic property accesses in Node.js programs [44], and found that it would at best have a small impact for our web-based benchmarks.

Our dynamic call graph and flow trace analyses were challenging to implement due to JavaScript's hard-to-analyze language features. JavaScript includes many difficult-to-analyze features, including (but not limited to) reflective call mechanisms, "native" library methods, getter/setter methods, and dynamic code evaluation. Pragmatic static analyses often ignore most of these features, as they do not aim for sound results. However, since we aimed to study which calls were missed by such analyses and *why* those calls were missed, our dynamic analyses had to faithfully capture the behavior of these features, and thereby incurred significant additional complexity (see Section 4.2).

All of our code and data is publicly available in an artifact [21]. Our infrastructure is reusable and could be applied to study other static analyses, other benchmarks, and other platforms (e.g., Node.js). Together, our infrastructure, methodology, and results can help guide the design of future analyses targeting real-world JavaScript code.

Contributions. This paper makes the following contributions:

- We present a novel approach to quantifying the importance of language features causing low recall in JavaScript call graphs. The approach properly handles missing call graph edges with multiple root causes, and also inter-dependent calls, where an edge is missing due to the absence of another edge.

- We describe implementations of a dynamic call graph and dynamic flow trace analysis of function values for JavaScript, both of which handle several hard-to-analyze JavaScript features.
- We present results and key observations from applying our techniques for the ACG algorithm [25] and a suite of framework-based web applications.

The remainder of this paper is organized as follows. Section 2 provides background, and Section 3 describes our dynamic analyses. Section 4 presents our technique for automatically discovering root causes for missing edges. Section 5 gives details of our implementation. Section 6 describes the setup of our study, and Section 7 presents our results. Section 8 discusses related work, and Section 9 concludes.

2 Background

We first give some background on challenges for JavaScript static analysis and on call graph construction.

2.1 JavaScript analysis challenges

JavaScript programs often pose particularly difficult challenges for static analysis. JavaScript includes numerous dynamic and reflective language features that are difficult to analyze, and unfortunately these features are used often in practice [49]. We briefly present such features here; see previous work for detailed discussions (e.g., [30, 46, 49, 55]). Tricky features include:

- **Dynamic Property Accesses:** JavaScript object fields, or *properties*, can be accessed using the syntactic form `x[e]`, where `e` is an arbitrary expression evaluating to a string property name. Determining what memory locations may be accessed by an expression `x[e]` (fundamental to tracking data flow) can be a significant analysis challenge. Further, if `e` evaluates to a property name that does not exist on `x`, a write to `x[e]` *creates* the property rather than failing, making precise analysis even more challenging.
- **Eval:** JavaScript allows for evaluating arbitrary strings as code at runtime, most commonly via its `eval` construct or the `Function` constructor. This dynamically-evaluated code is known to pose significant problems for static analysis [30, 48].
- **With:** The `with` construct enables adding arbitrary variable bindings with a dynamically-constructed map [2]. As with `eval`, `with` usage complicates static analysis [46].
- **Getters and Setters:** A JavaScript property may be defined such that accessing the property actually invokes a *getter* or *setter* method with custom logic [12]. This feature makes it difficult to precisely identify the program locations where a function call can occur.
- **Reflective Calls:** JavaScript provides reflective methods to pass function parameters in flexible ways, e.g., binding the `this` parameter explicitly or passing arguments in an array [13]. Also, any function may read its formal parameters via a special `arguments` array, enabling variadic functions. Finally, any function may be legally invoked with *any* number of parameters, independent of how many formal parameters it declares. Together, these features complicate tracking of inter-procedural data flow.
- **Native Methods:** JavaScript and the web platform provide a large standard library whose implementation is typically opaque to static analysis; hence, models must be constructed for a large number of these “native” methods.

While these root causes of difficult analysis are well known, our techniques enable measurement of their *relative* impact on call graph recall for a set of target benchmarks.

2.2 Call graph construction

In a static call graph, nodes represent program methods, and an edge from a to b means that a may invoke b at runtime.[2] The utility of a computed call graph CG can be measured in terms of *precision* and *recall*. Precision measures the number of infeasible edges in CG (edges for calls that cannot occur in any execution), while recall measures the number of feasible call edges (those that *can* occur in some execution) missing from CG. Recall will be 100% for any sound call graph construction technique, but as noted in Section 1, many practical techniques sacrifice soundness for improved scalability and precision. It is undecidable to compute the "ground truth" of possible calls for an arbitrary program, required to measure precision and recall perfectly. Our evaluation (and previous work [25, 44, 51, 57]) proceeds by exercising benchmarks using a best-effort process and then studying recall using the measured dynamic behaviors.

Static Flow Graphs. Our technique also relies on obtaining a *static flow graph* from the static call graph analysis, to determine what dynamic data flow of function values was missed by the static analysis (see Figure 1 and further discussion in Section 4). In a flow graph, each node represents either a memory location (variables, object properties, etc.), a function value, or a call sites. Edges in the flow graph are defined as follows: if the call graph analysis determines that a function value may be read from (abstract) memory location m_1 and then written to location m_2 (i.e., it may be directly copied from m_1 to m_2), the static flow graph should include an edge from m_1 to m_2. So, flow graph edges should capture observed assignments of function values into variables and object properties, and passing of function values as parameters or return values to capture inter-procedural data flow. Additionally, for a call $m_i(...)$, the flow graph should contain an edge from m_i to a "callee" node for the call site (see example below). With this construction, the static call graph should have an edge from call site s to function f iff there is a path from f to the callee node for s in the flow graph.

Graph representations are standard in analyses that track data flow [54]. Further, any realistic JavaScript call graph construction algorithm must track function data flow, as JavaScript provides no basis for a cheaper technique (functions cannot be coarsely matched to possible call sites using types or even function arity). Hence, we expect extraction of flow graphs from JavaScript call graph analyses will be straightforward.

Example. Figure 2 gives a small running example for illustrative purposes. Line 4 creates an object with two fields `MyName` and `MyPhone`, respectively holding functions `f1` and `f2`. Line 5 reads and invokes `f1` using a *static* property access (the property name is syntactically evident), whereas line 6 reads and invokes `f2` using a dynamic property access.

Figure 3 shows the flow graph constructed by a variant of the call graph builder we study [25] for the Figure 2 example. Edges represent the possible flow of function `f1` to the variable `v1`, then the object property `MyName`, and finally the call at line 5. Given this path, the static call graph includes an edge from `main` to `f1`. In contrast, the edge from the `MyPhone` property node to the call on line 6 is missing in Figure 3, due to the dynamic property access. Our approach can determine that this missing flow graph edge leads to a missing `main-to-f2` edge in the call graph, and further reason that a dynamic property access is the root cause of the missed edge.

[2] The call graph also includes information on which instruction in a, or *call site*, may invoke b.

```
1 function main() {
2   var v1 = function f1() { return "John"; }
3   var v2 = function f2() { return "555-1234"; }
4   var obj = { MyName: v1, MyPhone: v2 };
5   obj.MyName();
6   obj["My" + "Phone"]();
7 }
8 main();
```

Figure 2 Small example to illustrate our techniques.

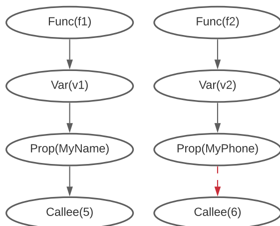


Figure 3 Flow graph for Figure 2. The red dashed edge is missing from the graph.

3 Dynamic Analyses

Our technique uses dynamic analyses to determine calls and data flows of function values occurring in executions of a program; this information is then compared with that in the static call graph and flow graph to detect missing flows (see Section 4). Here we describe the dynamic analyses at a high level; we discuss implementation challenges related to complex JavaScript language constructs (such as those listed in Section 2.1) in Section 5.

Dynamic Call Graphs. A dynamic call graph captures the calls that occurred in an execution (or set of executions) of a program. As with static call graphs, nodes represent program methods and edges represent invocations between methods. At a high level, constructing dynamic call graphs only requires recording the actual functions invoked at each call instruction in some suitable data structure, and this type of analysis has been built many times before, including for JavaScript [29]. However, our analysis goes further by exposing call-related behaviors of some of the tricky JavaScript constructs outlined in Section 2.1, crucial for a more complete understanding of static call graph recall.

Dynamic Flow Traces. Beyond dynamic call graphs, our technique requires *dynamic flow traces* to find gaps in the data flow reasoning of static call graph builders. A dynamic flow trace logs all data flow and invocation operations performed on function values. The trace includes an entry for each creation of a function value (e.g., an expression `function () { ... }`) and for each function call. It also includes an entry for each read or write of a function value to or from a variable or object property.

As an example, here is an excerpt of the dynamic flow trace for the code in Figure 2 (some details elided):

CREATE(`f1,2`); VARWRITE(`v1,f1,2`);
CREATE(`f2,3`); VARWRITE(`v2,f2,3`);
VARREAD(`v1,f1,4`); PROPWRITE(`MyName,f1,4`);

VarRead(`v2,f2,4`); PropWrite(`MyPhone,f2,4`);
PropRead(`MyName,f1,5`); Invoke(`f1,5`);
PropRead(`MyPhone,f2,6`); Invoke(`f2,6`);

Each entry includes information on the function value being accessed and the location of the access (here, line numbers). For property accesses, our traces only record the accessed property name, as the call graph techniques we studied in our evaluation do not distinguish base objects of accesses. The trace could easily be extended to include base object identifiers if needed to study other analyses.

For handling of higher-order functions, the trace includes entries for parameter passing and returns of function values. A call passing a function as a parameter is treated as a "write" of a parameter variable, which can be read via the formal parameter in the callee. For returns, a `return` statement "writes" a special variable associated with the function's return value, which is "read" at the corresponding call site.

4 Missing Flow Detection

In this section, we describe our technique for discovering the *missing flows* explaining why a static call graph is missing an observed dynamic call graph edge. See Figure 1 for our overall architecture. Given a dynamic flow trace for a program, we first post-process the trace to discover the relevant *dynamic copies* for a function call (Section 4.1). Then, our technique matches these dynamic copies to the static flow graph, and automatically computes the missing flows relevant to each missing call edge (Section 4.2).

4.1 Finding Relevant Dynamic Copies for a Call

Given a dynamic flow trace and an invocation of function f at a call site, our technique computes the *dynamic copies* by which f was invoked at the site. Dynamic copies capture data flow of function values at runtime – they are the dynamic analogue of the possible data flow captured in a static flow graph (Section 2.2). A dynamic copy captures one of three operations on function values: (1) the value is *created* and then stored in some memory location; (2) the value is *copied* from one memory location to another; and (3) the value is read from a location and *invoked*. By computing the relevant dynamic copies for a particular call, our technique can expose which data flows may have been missed by the static analysis.

Pseudocode for finding relevant dynamic copies appears in Algorithm 1. We use subscripted t variables for trace entries. Given an entry t_c for a call invoking function f in trace T, FindDynamicCopies computes a list C of the relevant dynamic copies, starting at the creation of f and ending at the call. Each dynamic copy is represented in the form $t_{r'} \xrightarrow{t_w} t_r$, read as: the function was read from memory by $t_{r'}$, and then copied to the memory location read by t_r, via write t_w. The algorithm proceeds *backwards* through the trace, starting at t_c and reconstructing step-by-step how f was copied through memory to reach the call site.

Algorithm 1 first finds the read or create operation t_r for f most closely preceding t_c in the trace (line 3), corresponding to evaluation of e in an invocation $e(\ldots)$.[3] C is then initialized with $t_r \xrightarrow{invoke} t_c$, with the *invoke* label indicating this is not a true copy, but instead the invocation of f.

[3] In certain corner cases, the closest preceding operation may not be the correct one; we discuss further under Limitations.

Algorithm 1 Finding dynamic copies for a call.

1: **procedure** FindDynamicCopies(T, t_c)
2: $f \leftarrow$ function invoked by t_c
3: $t_r \leftarrow$ PrecedingReadOrCreate(T, t_c, f)
4: $C \leftarrow [(t_r \xrightarrow{invoke} t_c)]$
5: **while** t_r is not a Create operation **do**
6: $t_w \leftarrow$ MatchingWrite(T, t_r, f)
7: $t_{r'} \leftarrow$ PrecedingReadOrCreate(T, t_w, f)
8: $C \leftarrow (t_{r'} \xrightarrow{t_w} t_r) :: C$
9: $t_r \leftarrow t_{r'}$
10: **end while**
11: **return** C
12: **end procedure**
13: **procedure** MatchingWrite(T, t_r, f)
14: **if** t_r reads variable x **then**
15: **return** PrecedingVarWrite(T, t_r, f, x)
16: **else if** t_r reads property *prop* **then**
17: **return** PrecedingPropWrite($T, t_r, f, prop$)
18: **else if** t_r reads formal p of function f' **then**
19: // *preceding invoke of f' passing f to p*
20: **return** PrecedingInvoke(T, t_r, f', f, p)
21: **else if** t_r is return value of call to f' **then**
22: // *preceding return of f from f'*
23: **return** PrecedingReturn(T, t_r, f', f)
24: **end if**
25: **end procedure**

The loop at lines 5–10 discovers relevant dynamic copies by matching writes and reads backward in the trace. First, Algorithm 1 finds the closest-preceding write operation t_w that updated t_r's location, using the MatchingWrite procedure. MatchingWrite's logic proceeds in cases, handing variables, object properties, formal parameters, and return values in turn. For a read of property *prop*, the pseudocode matches with the most recent write to *prop* on any object, matching the heap abstraction used by the call graph builder variants we study (see Section 6.1). For more precise call graph algorithms, the logic could easily be updated to also match the exact base object used in the property read operation. Once the matching write t_w is discovered, line 7 finds the closest-preceding read or create $t_{r'}$, which "produced" f for the write, and prepends a dynamic copy $t_{r'} \xrightarrow{t_w} t_r$ to C.

As an example, consider the call to `f2` on line 6 in Figure 2. Here are the relevant trace entries for that call visited by Algorithm 1:

Create(`f2,3`); VarWrite(`v2,f2,3`);
VarRead(`v2,f2,4`); PropWrite(`MyPhone,f2,4`);
PropRead(`MyPhone,f2,6`); Invoke(`f2,6`);

Starting from the Invoke entry, the closest preceding read of `f2` is the PropRead of `MyPhone` on line 6. So, C is initialized with PropRead(`MyPhone,f2,6`) $\xrightarrow{invoke}$ Invoke(`f2,6`). The matching PropWrite for the read occurs on line 4, and its closest preceding read of `f2` is the VarRead on line 4. Hence, we prepend a dynamic copy VarRead(`v2,f2,4`) $\xrightarrow{t_{w_1}}$ PropRead(`MyPhone,f2,6`), where t_{w_1} = PropWrite(`MyPhone,f2,4`). Proceeding similarly, we reach the creation point of `f2` on line 3, prepend a dynamic copy Create(`f2,3`) $\xrightarrow{t_{w_2}}$ VarRead(`v2,f2,4`), where t_{w_2} = VarWrite(`v2,f2,3`), and terminate.

Limitations

Algorithm 1 assumes that the most-closely-preceding read of a function f in the trace matches the subsequent write or invocation involving f. In rare cases with parameter passing, this assumption may not hold, e.g.:

```
1  function foo(p, q) { p(); }
2  function bar() {}
3  var x = bar;
4  var y = bar;
5  foo(x, y);
```

Assume we are trying to discover the dynamic copies for the call to `bar` on line 1. Here is the relevant excerpt of the flow trace:

> ...; VarWrite(`x`,`bar`,`3`); VarWrite(`y`,`bar`,`4`); VarRead(`x`,`bar`,`5`);
> VarRead(`y`,`bar`,`5`); Invoke(`foo`,`5`); VarRead(`p`,`bar`,`1`); Invoke(`bar`,`1`);

For the final Invoke of `bar`, the closest-preceding read is of formal parameter `p`. The matching "write" is the Invoke of `foo` on line 5. From here, the closest-preceding read of `bar` is from variable `y`, which is *not* the parameter that gets passed in `p`'s position. Hence, the analysis will discover an infeasible dynamic copy from the read of `y` to the read of `p`. This simple case could be handled by using source locations during matching, but in cases involving recursion, dynamic call stacks would also need to be tracked. We did not observe this behavior in any of our benchmarks, so we chose to employ the simpler technique of Algorithm 1.

In some cases, the dynamic flow trace may be missing entries relevant to dynamic copies, due to JavaScript features like native methods and `with` (Section 2.1) and also implementation limitations; see Section 5 for details. In such cases, our algorithm returns the subset of the relevant dynamic copies that it is able to reconstruct, and if possible notes a reason for its failure to find all copies.

4.2 Flow Graph Matching

Given relevant dynamic copies for a call c missed in the static call graph (discovered based on comparison with the dynamic call graph), we identify the missing flows for c by matching the dynamic copies to the static flow graph extracted from the call graph builder. (Section 2 described static flow graphs, and Figure 3 gave an example.) Algorithm 2 gives pseudocode for finding missing flows in a static flow graph. The routine FindMissingFlows takes as inputs a list of dynamic copies C produced by FindDynamicCopies in Algorithm 1, a static call graph CG, and the corresponding static flow graph FG. Its result is a set of missing flows R, where each missing flow is one of three types: (1) MissingFGNode, indicating a node is missing in the flow graph, (2) MissingFGPath, indicating a path is missing in the flow graph, and (3) DependentCall, for when the absence of a flow is due to the absence of another call in the call graph.

For a dynamic copy $t_{r'} \xrightarrow{t_w} t_r$, the algorithm first tries to identify corresponding flow graph nodes *fgSrc* and *fgDst* (lines 4 and 5). In most cases, this matching is straightforward, done either by matching code entities or matching an accessed memory location to the flow graph node that abstracts it (we elide the details). In some cases, the flow graph may not have a matching node, e.g., due to use of `eval` or due to an unmodelled property name from a dynamic property access. In such cases, we record an MissingFGNode entry in the result (lines 6–11).

If flow graph nodes *fgSrc* and *fgDst* are discovered, we next check for a *path* from *fgSrc* to *fgDst* in the flow graph (line 12). We must check for a path, rather than just an edge,

Algorithm 2 Finding missing flows in a flow graph for a call.

```
1: procedure FindMissingFlows(C, CG, FG)
2:     R ← ∅
3:     for each dynamic copy t_{r'} --t_w--> t_r ∈ C do
4:         fgSrc ← FlowGraphNode(FG, t_{r'})
5:         fgDst ← FlowGraphNode(FG, t_r)
6:         if fgSrc = null then
7:             R ← R ∪ MissingFGNode(t_{r'})
8:         end if
9:         if fgDst = null then
10:            R ← R ∪ MissingFGNode(t_r)
11:        end if
12:        if fgSrc ≠ null ∧ fgDst ≠ null ∧ NoPath(FG, fgSrc, fgDst) then
13:            R ← R ∪ MissingFGPath(fgSrc, fgDst, t_{r'}, t_w, t_r)
14:        end if
15:        if t_w is a call then
16:            f ← function invoked by t_w
17:            if MissingFromCG(CG, t_w, f) then
18:                R ← R ∪ DependentCall(t_w, f)
19:            end if
20:        end if
21:    end for
22:    return R
23: end procedure
```

since the static analysis may use temporary variables and assignments not present in the source code. If no path is discovered, we note a MissingFGPath entry, retaining information about the dynamic copy to facilitate root cause labeling.

As an example, consider again the call to `f2` in Figure 2, and the corresponding dynamic copies described in Section 4.1. In the Figure 3 flow graph for the code, there are matching nodes for all the copy locations, but there is no path matching the final copy PropRead(`MyPhone,f2,6`) $\xrightarrow{invoke}$ Invoke(`f2,6`). So, the single missing flow computed for this case is a MissingFGPath entry with the details of this dynamic copy. Given this information, a root cause labeler can discover that the flow was missed due to the dynamic property access; see Section 6.2.

Dependent calls. Lines 15–20 handle *dependent calls*, where a path corresponding to a parameter passing or return dynamic copy is missing from the flow graph due to *some other* missed call. Consider this example:

```
1 function f() { ... }
2 var x = { foo: function f2() { return f; } };
3 var y = x["fo"+"o"]();
4 y();
```

For the optimistic ACG call graph algorithm we use in our evaluation (see Section 6.1), the calls to `f2` at line 3 and to `f` at line 4 will be missing in the call graph. When finding missing flows for the line 4 call, a missing path for the function return dynamic copy at line 3 is discovered. However, the issue with the analysis is not that it does not model returns of function values; this flow was missed *because* the call target at line 3 was missed, so no flow could be discovered from the appropriate callee. Our discovery of missing flows must account for such cases, to enable accurate quantification of root causes.

To handle dependent calls, Algorithm 2 checks at line 15 if the "write" operation for the copy was a call. (Recall from Section 3 that calls are treated as the writes for parameter passing or function returns.) If so, and if the static call graph is missing the relevant target for the call (line 17), we add a DependentCall missing flow to the result (line 18).

When counting the frequency of root causes, for dependent calls, we *reuse* the root causes for one call as the root causes for the other. For the example above, the dynamic property access at line 3 is identified as the single root cause for the missing calls at lines 3 and 4. All results presented in Section 7 precisely account for dependent calls.

Root Cause Labeling. Given a set of missing flows, quantification of root cause prevalence requires attributing a *root cause label* to each missing flow. The root cause labels may be specific to the call graph construction algorithm being studied, and must be developed with knowledge of the soundness gaps in the algorithm. Additionally, root cause labeling may be performed with different levels of granularity, depending on what information is required by the analysis developer. In Section 6.2, we discuss the root cause labeling strategies used in our example study of the ACG call graph algorithm [25].

5 Implementation

Dynamic analyses. We implemented our dynamic call graph (DCG) and dynamic flow trace analyses (Section 3) atop the Jalangi framework [53],[4] which leverages source code instrumentation. While this instrumentation approach is more maintainable and portable than the alternatives, a downside is that the semantics of certain language constructs are not exposed in a straightforward way at the source level. In spite of source code instrumentation's limitations, one of its primary advantages is that it does not require modification of a JavaScript engine. Production JavaScript engines in browsers are challenging to modify, for two reasons: (1) they have complex implementations, so any change will require considerable engineering effort; and (2) they evolve rapidly, making it difficult to maintain an analysis. We use Jalangi2 to instrument JavaScript programs with our analysis code because it is easy to maintain and can work across different JavaScript engines. The tool allows us to perform analyses even when certain fragments of the source code are not instrumented. Our analyses contain significant extra logic to capture the behavior of several hard-to-analyze constructs as accurately as possible, despite the limitations of source instrumentation.

As an example, our DCG analysis exposes many callbacks from "native" library functions. Such callbacks occurred regularly in the benchmarks used in our study, e.g., using `Function.prototype.call`, as shown in this small example:

```
1    function foo() { }
2    foo.call(this);
```

Line 2 invokes `foo` via `call`, but Jalangi does not expose the invocation directly, as it cannot instrument `call`. Instead, Jalangi exposes the invocation of `call`, followed by the start of execution in `foo`, but with no explicit invocation of `foo`. To handle such cases, our DCG analysis maintains its own representation of the call stack. Upon invocation of a native method, a marker for the method is pushed on the call stack. Then, at the entry of a (non-native) method, if the top of our call stack is a native method marker, we record the

[4] We use version 2 of Jalangi, available at `https://github.com/Samsung/jalangi2`.

fact that a native callback occurred. For the above case, the dynamic call graph will include an invocation of the `call` native method at line 2, and also an invocation of `foo` from `call`, as desired.[5]

Our DCG analysis also exposes getter and setter calls, and calls to and from dynamically-evaluated code. For getters and setters, the analysis detects their presence via a library API [1]. If a getter or setter is detected at a property access, it is treated as a call site and the call edge is recorded. We leverage Jalangi's built-in support for dynamic code evaluation via `eval` or `new Function`; the relevant code string gets instrumented at runtime, so our analysis has visibility into calls into or out of such code.

Our dynamic flow trace analysis also includes special handling of some challenging JavaScript features. The analysis distinguishes getters and setter calls using specially-marked Invoke entries, to enable tracking getter and setter use as a root cause. For uses of the `arguments` array to access parameters, we generate relevant property write entries at a function entry as "synthetic" entries (not corresponding to explicit source code). To handle `eval`-like constructs, any trace entry from the evaluated code includes a special source location marking it as from code executed via `eval`.

JavaScript has a very broad set of features and native methods requiring special handling, and our dynamic analyses still do not model all such features. For the flow trace analysis, in certain cases a property write or read occurs in an unmodelled native method, and hence is missed in the trace. The analysis generates special entries to model memory accesses performed by commonly-used library methods, such as `push` and `pop` on arrays. We have not fully modeled all reflective constructs like `Object.defineProperty` [14]. Also, use of the `with` construct can thwart our technique, as it is not fully supported by Jalangi. (We note that all relevant uses of `with` in our benchmarks appeared *within* an eval construct,[6] posing a severe challenge for static analysis.)

In terms of performance, we implemented some optimizations to reduce the size of the dynamic flow trace for larger benchmarks. First, we limited tracing to only those function values that could be involved in a missing edge in the static call graph, based on the creation site of the function. Second, we track a unique identifier for each function value using Jalangi's shadow memory functionality, and once the call site with the missing static call graph edge executes, we disable flow tracing for the corresponding value.

To generate dynamic call graphs and flow traces, we exercised our benchmarks manually and recorded the actions as Puppeteer [15] automation scripts to allow for repeatable runs; Section 6.3 details the coverage obtained for benchmarks in our study.

Missing Flow Detection. The missing flow detection algorithms of Section 4 are implemented in `1154` lines of Python code. For the most part, detecting missing flows in the static flow graph given a dynamic flow trace was straightforward. Some effort was required to match source locations provided by WALA [58] for JavaScript constructs (our use of WALA is detailed in Section 6.1) with what was observed by the dynamic analyses. In the process of ensuring this matching was precise, we contributed a couple of fixes to WALA, and also found and fixed a longstanding issue with incorrect source locations in the Rhino JavaScript parser [5].[7]

---

[5] Our technique does not yet precisely handle cases with multiple levels of native calls, such as `Array.prototype.map.call(...)`; we plan to add further modeling for such cases in the future.
[6] For example, see this code from the Knockout framework: https://tinyurl.com/1jxtrpz3
[7] https://github.com/mozilla/rhino/pull/809

6 Study Setup

Here, we detail the setup of our study of root causes of missed call graph edges for framework-based web applications. We describe the ACG call graph algorithm used in our study (Section 6.1), describe how we performed root cause labeling for this algorithm (Section 6.2), and then present our benchmarks and how they were exercised (Section 6.3).

We note that the main purpose of our study was to show the potential of our techniques to give useful insights on the relative importance of different root causes for missed static call graph edges. We do *not* claim that the results for the benchmarks used in our study will generalize to any broad class of framework-based web applications. A study of a wider variety of benchmarks, to obtain generalizable insights on root causes across JavaScript applications, is beyond the scope of this work.

6.1 The ACG algorithm

In our evaluation, we studied variants of the approximate call graph (ACG) algorithm of Feldthaus et al. [25]. The ACG algorithm was designed to entirely skip analysis of many challenging JavaScript language features, while still providing good precision and recall for real-world programs. ACG leverages the insight that many dynamic property accesses in JavaScript are correlated [55], with a paired dynamic read and write used to copy a property from one object to another. By using a *field-based* handling of object properties [28] (treating each property as a global variable), ACG could ignore dynamic property accesses entirely and still provide good recall, assuming most accesses are correlated.

Feldthaus et al. [25] describe *pessimistic* and *optimistic* variants of ACG, differing in their handling of inter-procedural flow. Pessimistic ACG only tracks data flow across procedure boundaries in limited cases, whereas optimistic ACG performs full inter-procedural tracking. We performed root cause quantification for both variants in our study.

Our study uses the open-source implementation of ACG in WALA [58]. This implementation directly builds a flow graph during call graph building, which we serialize alongside the computed call graph. The WALA implementation also includes partial handling of the `call` and `apply` reflective constructs for parameter passing [13]. In the optimistic variant, interprocedural flow is handled fully for `call`, but only return values are handled for `apply` (as it passes parameters via arrays, which is hard to analyze). We confirmed via inspection that the WALA implementation of ACG has no handling of getters and setters, `eval`, and `with`.

6.2 Root Cause Labeling

We implemented root cause labeling for missing flows based on the gaps we observed in the WALA implementation [58] of the ACG algorithm [25]. For a different algorithm or implementation, some different root causes may be required, but we expect significant overlap, as several root causes pertain to challenging language features that many techniques handle unsoundly (e.g., `eval`). The referenced root cause names are also used when discussing their prevalence in Section 7.2.

For MissingFGNode (see Section 4.2), in some cases, there is no node representing the creation of a function value in the flow graph. If the function was from the standard library, we assigned the label "Call to unmodelled native function," as WALA was likely missing a model for the function. In cases where the function was created via a call to `new Function` (unhandled by the ACG implementation), we assigned the label "Creation via Function constructor."

In other MissingFGNode cases, the node representing the call site itself is missing. For this case, a common root cause label is "Call to getter/setter," as getters and setters are not modeled by ACG. Also, the "Calls from unmodelled native functions" label captures cases where an unmodeled native function calls back into application code. Finally, for a dynamic property access, if the property name is never used as part of a non-dynamic property access, the flow graph may not have a node for the property, in which case we use the label "Dynamic Property Access."

For MissingFGPath, one possible root cause is "Dynamic Property Access," which can be identified by the corresponding dynamic reads / writes. For the pessimistic ACG variant, paths may be missing since the algorithm does not model passing function values as parameters or returning function values; we use the labels "Parameter Pass" and "Function return" for these scenarios. For both ACG variants, the "Parameter Pass" label is also used to reflect passing of parameters in an array via `Function.prototype.apply`.

In the case of dynamically-evaluated code (the "Use of Eval" and "Eval via new Function" labels), many relevant nodes may be missing from the static flow graph. We assign an appropriate root cause in these cases by recording in the flow trace which events occurred in dynamically-evaluated code (Section 5). Note that we *prioritize* the `eval`-related root causes over others; e.g., if there is a relevant dynamic property access in `eval`'d code, we will assign the `eval`-related root cause, even though it is possible the analysis also could not handle the property access. We chose this labeling due to the high difficulty of handling `eval` constructs in static analysis; for an analysis with significant support for `eval` a different choice may be appropriate.

Finally, as noted in Section 4.1, in certain cases we cannot compute all dynamic copies for a call. For these cases, our technique makes a base-effort attempt to assign an appropriate root cause label. "Call to bounded function" captures missing handling of the `Function .prototype.bind` feature [13]. The "Multiple levels of native functionality" label captures cases where native methods are invoked reflectively (see Footnote 5). Finally, we identify the "Use of With" root cause by tracing objects used in `with` statements and identifying when an unmatched variable corresponds to a `with` object property.

As Section 7.2 will show, dynamic property accesses are the most frequent root cause of missing call graph edges for our benchmarks. To further understand these root-cause accesses, we also implemented a finer-grained labeling for them, based on the expression used for the property name. This more granular labeling is described in Section 7.3.

6.3 Benchmarks and Harness

For benchmarks, our study used several programs from the TodoMVC suite [17]. TodoMVC contains many implementations of a simple web-based todo list application, with each implementation using a different web framework or language. The suite is designed to help developers compare different model-view-controller (MVC) frameworks. Because the suite contains idiomatic implementations of the same functionality across frameworks, it provides an opportunity to compare sources of missing call graph edges across frameworks.

To test with a larger web application, we also included OWASP Juice Shop [3], an AngularJs-based program that is a standard benchmark for security analyses. Counting the size of framework / library code for Juice Shop is difficult, as the code base does not clearly separate third-party code used as part of the web site from libraries used only to deploy the site; we conservatively estimated the framework / library code to be greater than 50 kLoC.

Table 1 Benchmark Statistics.

	Total LoC	Application LoC	Framework/ Library LoC	Application Stmt. Coverage
AngularJs	12091	256	11835	81.08%
Backbone	9003	216	8787	99.74%
KnockoutJs	1044	129	915	98.98%
KnockbackJs	15836	199	15637	99.73%
CanJs	11371	129	11242	100%
React	24855	383	24472	99.21%
Mithril	1433	252	1181	99.61%
Vue	7667	124	7543	97.73%
VanillaJs	751	561	190	98.10%
jQuery	9526	171	9355	99.59%
Juice Shop	>65000	15092	>50000	36%

Table 1 gives statistics for our benchmarks. The TodoMVC benchmarks are named based on the web framework that they use. The TodoMVC applications range from 751–24,855 lines of code, with framework sizes varying widely. We chose all eight of the JavaScript-framework-based implementations that worked with our infrastructure.[8] We also chose VanillaJS, which does not use any framework,[9] and jQuery, for comparison purposes.

To exercise the TodoMVC applications, we wrote a harness to cover as much application code as possible, and in the end our script achieved application code statement coverage of 97% or higher for nearly all benchmarks. We studied all uncovered code manually, and found that it was either dead code or could not be exercised in a single run of the application (e.g., for the AngularJs version, a small amount of code would only run if the app were used and then restarted in offline mode).

For Juice Shop, we were unable to exercise the application beyond fully completing its initial loading, explaining the significantly lower code coverage. Our infrastructure ran into scalability issues for deeper runs of Juice Shop, which we hope to fully address in the near future. Still, simply loading Juice Shop exercised a large amount of code (its flow trace was nearly 5 times larger than any fully-exercised TodoMVC benchmark), making a study of missed call edges for the loading portion of the execution interesting on its own.

In terms of running times for our tools, dynamic call graph and flow trace collection each took between 30 and 60 seconds for each TodoMVC benchmark, varying based on the amount of code executed; this overhead is comparable to previous Jalangi-based dynamic analyses [53]. Missing flow detection (Section 4) took time proportional to the size of the flow trace, ranging from around half a second (for VanillaJS) to around 10 minutes (for React). Overall running time for Juice Shop was much longer (more than an hour total) due to its size and the aformentioned scalability bottlenecks it exposed. We expect the missing flow detection times could be reduced significantly with a more optimized implementation.

[8] Some implementations used newer JavaScript language features not yet supported by Jalangi.
[9] All implementations use a common base JavaScript library, accounting for the library code in VanillaJS.

7 Results

In this section, we present results from performing root cause quantification for our benchmarks. The results show that our quantification techniques can provide interesting insights into the relative prevalence of different root causes for missing call graph edges. We first give recall measurements for our benchmarks using multiple metrics in Section 7.1. Then, we discuss the top root cause labels for missed call graph edges in Section 7.2 and insights gained from this data. Finally, we discuss results from performing a finer-grained labeling of missing flows related to dynamic property accesses (the most prevalent root cause) in Section 7.3.

7.1 Recall Measurements

We measured call graph recall for our benchmarks by comparing the ACG static call graphs with our collected dynamic call graphs. We first describe our methodology, and then present results. We also measured call graph precision for all benchmarks, but as our new techniques focus on root causes for low recall, we do not discuss the precision results here; they are presented in an extended version of the paper [22].

Methodology. We used three different metrics to measure recall, suited to different client scenarios:

- **Call site targets:** the set of targets at each call site present in the dynamic call graph. This metric was used in the original ACG paper [25]. Recall is computed for each call site, and then averaged across call sites to produce recall for a benchmark. This metric is most relevant to clients like code navigation in an IDE.
- **Reachable nodes:** the set of reachable methods, where roots are the entrypoints in the dynamic call graph. This metric has been used in previous work [57], and is relevant to clients like dead-code elimination.
- **Reachable edges:** the set of call graph edges whose source method is present in the dynamic call graph. This metric is most relevant to clients doing deep inter-procedural analysis like taint analysis [26].

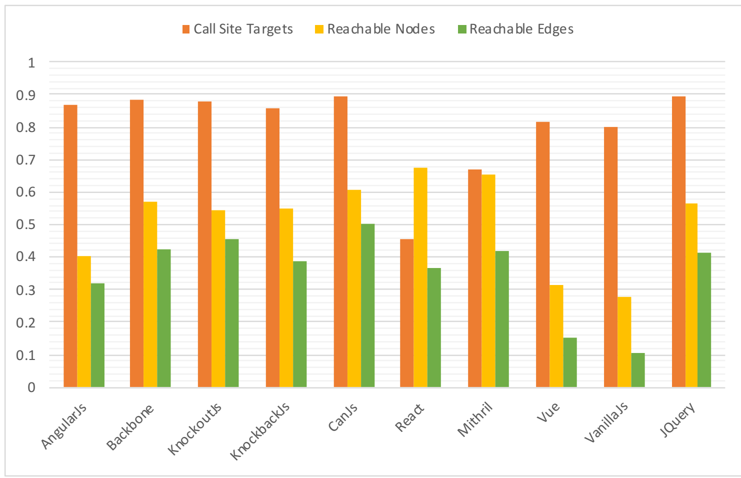
Given our collected data, we studied the following research questions:

- **RQ1:** How does recall vary across the three metrics?
- **RQ2:** How does recall vary across benchmarks?

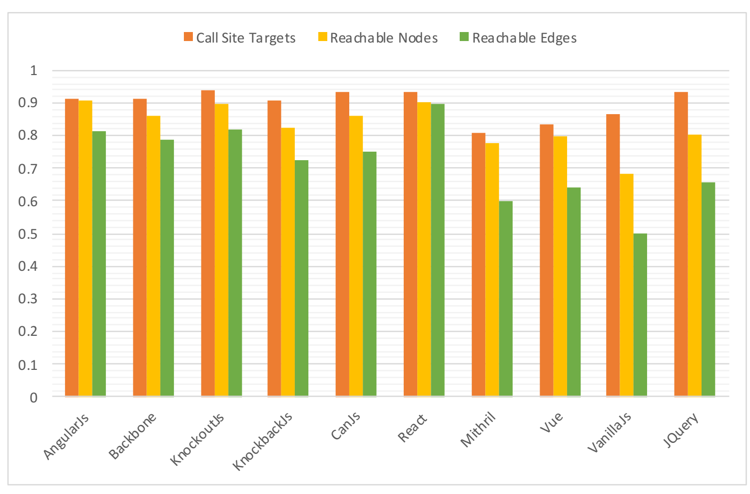
Results. Figure 4 gives detailed recall results for WALA's original ACG implementation for each TodoMVC benchmark, with results for the pessimistic variant in Figure 4a and for optimistic in Figure 4b. Average recall across the TodoMVC benchmarks is shown in Figure 5.

For RQ1, the data show that recall of ACG tends to suffer with more exacting metrics. The ACG paper [25] used the call site targets metric, and showed that both precision and recall were typically above 80% for their benchmarks. Figure 5 shows that for our benchmarks, while recall is above 80% for this metric for both the optimistic and pessimistic variants, recall decreases for the more exacting metrics, particularly for pessimistic analysis.

For RQ2, Figure 4 shows that recall can vary widely across benchmarks. In Section 7.2 we dig further into these differences, showing that root causes for low recall can also vary across the benchmarks. For the TodoMVC React benchmark, recall is very high for the optimistic analysis but quite low for pessimistic. In this case, the high recall for optimistic



(a) Pessimistic ACG.



(b) Optimistic ACG.

Figure 4 Detailed recall results for our three metrics across the benchmarks.

analysis comes at a cost of very low precision (less than 5% for reachable edges; see the extended version of the paper [22] for full details). We suspect that some initial imprecision spirals out of control for optimistic analysis for React, leading to poor precision. Previous work studied diagnosing imprecision root causes [20, 35, 60]; such a study is out of scope here. However, improving recall can lead to reduced precision, and this tradeoff must be minded when devising solutions to improving recall.

For Juice Shop, only the pessimistic ACG variant could run to completion; optimistic ACG could not complete within 64GB of memory. Pessimistic ACG missed 15,060 edges that were present in the dynamic call graph. Since our coverage for Juice Shop was significantly lower than the other benchmarks (see Section 6.3), we do not quantify the precision and recall of pessimistic ACG for the benchmark, nor do we include it in aggregate statistics.

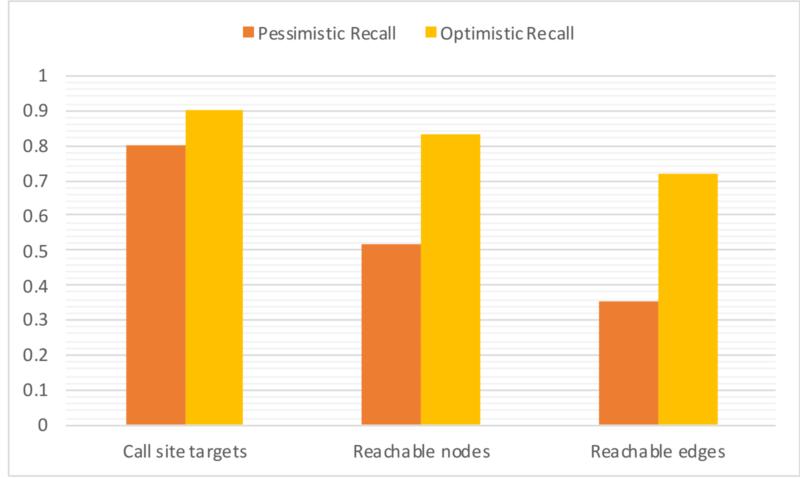


Figure 5 Average recall across benchmarks for original WALA ACG implementation.

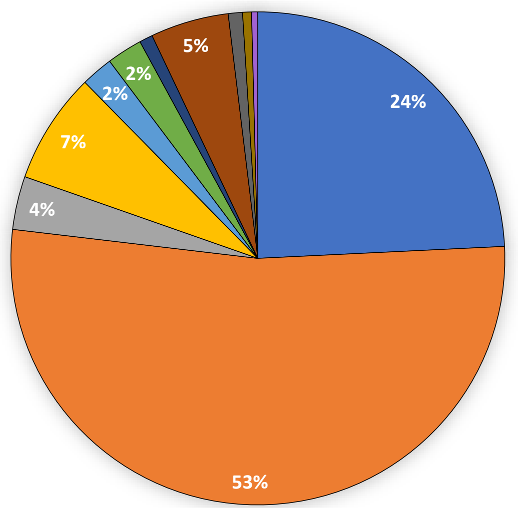


Figure 6 Original root causes for optimistic ACG across TodoMVC, before WALA improvements.

7.2 Root Cause Quantification

We present illustrative results from applying our techniques to quantify prevalence of root causes for missing call graph edges for our benchmarks. Space does not allow a full presentation of all results; all experimental data is available in our artifact [21]. Here we focus on the following questions:

- **RQ3:** What are the most common root causes for missed call graph edges?
- **RQ4:** Does the relative importance of root causes vary across benchmarks?

We compute root causes for each individual missed call edge in the static call graph, corresponding to the "Reachable edges" metric used to measure recall in Section 7.1. The color legend for the pie charts appears below Figure 8.

Using data to improve recall. Figure 6 shows the prevalence of different root causes across the TodoMVC benchmarks for the optimistic variant of the original ACG implementation in WALA. When studying these root causes, we were surprised to see that 24% of missed call edges were due to calls to unmodeled standard library functions. Based on this data, we modified WALA to include basic models of many of these native functions. This change improved average recall for the pessimistic analysis by 2 percentage points to 37% (by the Reachable Edges metric); improvement for optimistic analysis was 5 percentage points, to 76%. These improvements show that quantifying root cause prevalence can guide an analysis developer to "quick wins" for improving analysis recall. The data in the remainder of this section were computed using the improved version of WALA ACG.

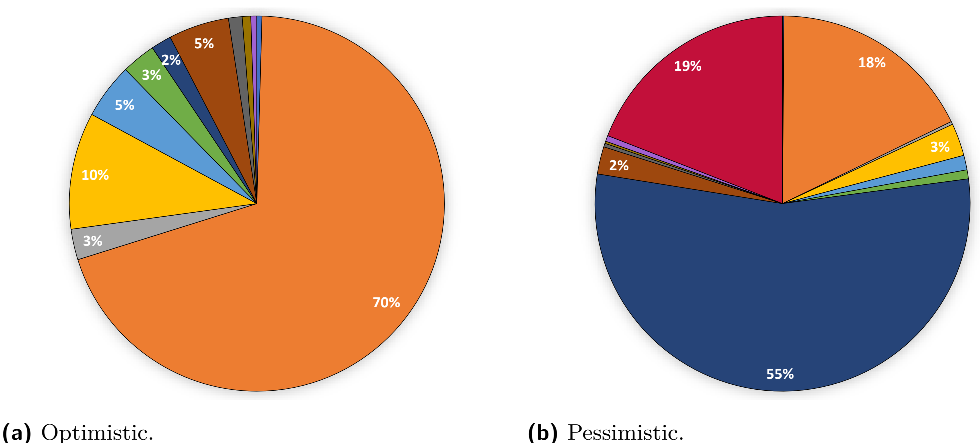
Top root causes. Turning to RQ3, Figures 7a and 7b respectively show top root causes for pessimistic and optimistic ACG across the TodoMVC benchmarks (after improving WALA's native models). Comparing the two, we see a key difference is that missed calls due to functions being passed as parameters or returned (the "Parameter Pass" and "Function return" labels) are significant root causes (totaling 74%) for pessimistic analysis but not optimistic. This result makes sense, as the key difference between optimistic and pessimistic ACG is that optimistic analysis tracks interprocedural flow of function values. Given that 74% of missed edges for pessimistic analysis are due to such interprocedural flows, it seems the best approach to improving pessimistic recall for these benchmarks would be to model some of these flows, rather than attacking other root causes.

The "Others" label covers a small number of cases (5% overall) where our current scripts cannot yet find a root cause. In addition to the unhandled constructs and cases described in Section 5, our automated reasoning failed in rare cases due to a bug in WALA ACG's handling of `finally` blocks. During our work, we identified two other WALA ACG bugs that were fixed by the maintainers. Overall, our techniques successfully handle more than 95% of the missing call edges for our benchmarks, and we will continue to improve our tools to reduce the number of unhandled cases.

Focusing in on Figure 7a, we see that dynamic property accesses are by far the most prevalent root cause for optimistic analysis of TodoMVC benchmarks at 70%. We dig further into these property accesses with a finer-grained labeling in Section 7.3. The second-most prevalent root cause on average is "Eval via new Function" at 10%, but as we shall see next, the second-highest root cause varies significantly across benchmarks.

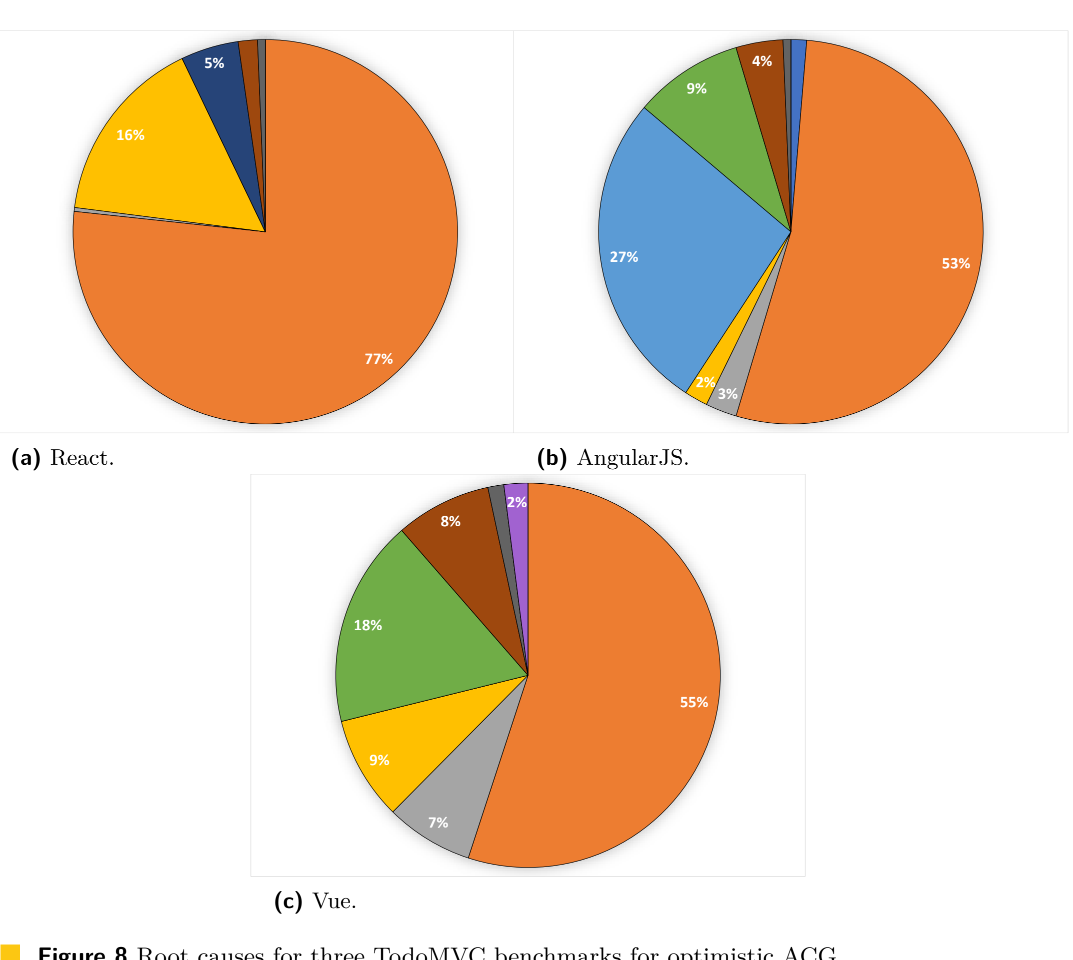
Variance across benchmarks. For RQ4, we use illustrative examples to show the variance in root cause prevalence across benchmarks. Figures 8a–8c respectively show root causes for the React, Angular, and Vue.js TodoMVC benchmarks, analyzed with optimistic ACG. While the most-prevalent root cause for each of these benchmarks was dynamic property accesses, the second-place root cause varies by benchmark: "Eval via new Function" is second for React, "Call to bounded functions" for AngularJS, and "Call to getter / setter" for Vue. This benchmark-specific data could provide valuable information to an analysis developer. E.g., if the developer were primarily trying to improve recall for applications like the Vue benchmark, it may be more worthwhile to improve handling of getters and setters than if the applications were more similar to the React benchmark.

Figure 9 shows root causes for the larger Juice Shop benchmark (analyzed with pessimistic ACG). Unfortunately, Juice Shop exercised gaps in our infrastructure's handling of tricky JavaScript constructs more heavily, particularly in the dynamic flow trace analysis. So, we could not compute proper root causes for 27% of missing call graph edges for Juice Shop.



(a) Optimistic. **(b)** Pessimistic.

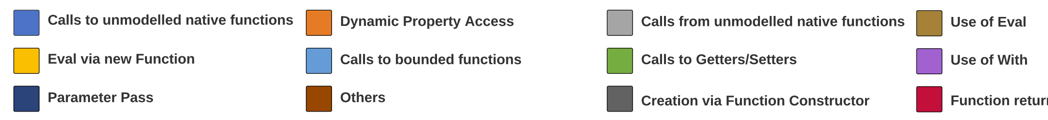
Figure 7 Improved root causes for ACG variants across TodoMVC, after WALA improvements.



(a) React. **(b)** AngularJS.

(c) Vue.

Figure 8 Root causes for three TodoMVC benchmarks for optimistic ACG.



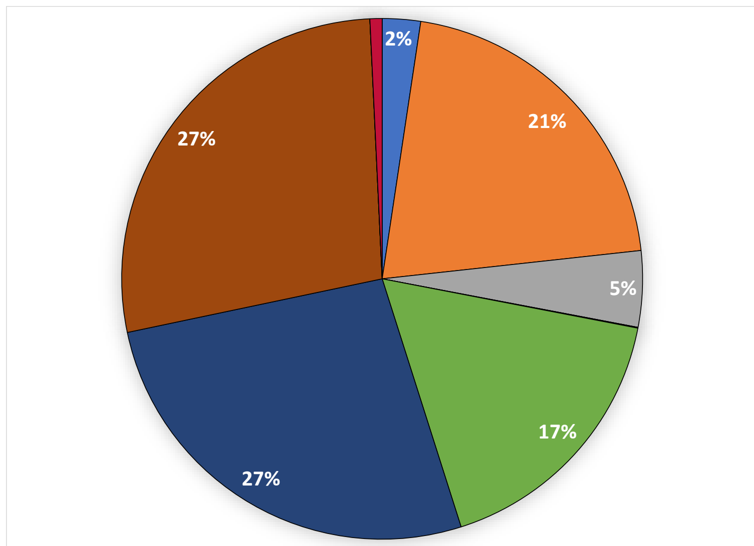


Figure 9 Root causes for pessimistic ACG for Juice Shop.

Still, the remaining data is interesting, particularly when compared to the pessimistic results for the TodoMVC benchmarks shown in Figure 7b. We see that handling returns of functions seems to be relatively less important than for the TodoMVC benchmarks, whereas handling of getters and setters is more important. Though making strong conclusions is difficult given the number of uncategorized edges in this case, these preliminary data again show the ability of our technique to expose benchmark-specific insights about causes of low recall.

To summarize, we have shown that our technique for quantifying root causes works across several benchmarks and can expose the most important root causes in aggregate and the differences between benchmarks. Since improving recall for JavaScript static analysis on real-world programs poses so many challenges, we expect improvements for specific types of benchmarks to prove worthwhile, and the data from our techniques can provide valuable guidance in how to do so.

7.3 Name Flow for Dynamic Property Accesses

Given the importance of dynamic property accesses as a root cause in Section 7.2, we performed a finer-grained root cause labeling of these accesses. Our goal was to understand better how property names are computed for these accesses, to see if some targeted handling of the property name expressions could be useful. Recent work by Nielsen et al. [44] proposes just such a technique for analysis of Node.js code, via special handling of property name expressions that concatenate a string constant prefix or suffix to some other expression. We hoped to use root cause labeling to see if a similar technique could be effective for our web-based benchmarks.

We implemented a simple intra-procedural analysis using WALA [58] to label each root-cause dynamic property access based on how data flows into its property name expression (for an access `x[e]`, `e` is the property name expression). Aggregate results appear in Figure 10; our artifact has the complete data [21]. As shown in Figure 10, property names for root-cause dynamic accesses have a diverse set of sources. The largest single source are JavaScript's `for-in` loops for iterating over object properties, studied frequently in the literature as a challenge for static analysis (e.g., [19, 47]). However, they account for only 31% of cases in

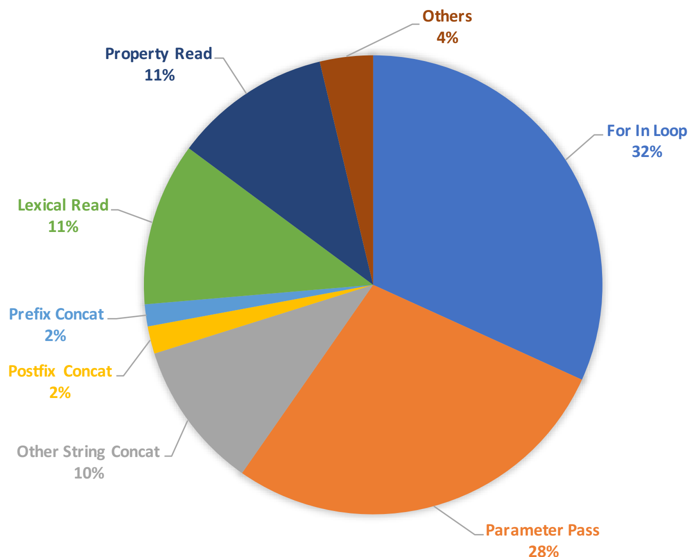


Figure 10 Finer-grained dynamic property access root causes for TodoMVC benchmarks.

total, and many other sources exist. Property names are often passed in from outside the function containing the access, whether by parameter passing (28%) or variables in enclosing lexical scopes (12%); handling these cases may require inter-procedural tracking of property name value flow. Another major source is property reads (12%) (i.e., the property name is read from another object property), whose handling may again require deep tracking of value flow.

String concatenation cases comprise 14% of root-cause property name expressions. Only 4% of such expressions in our benchmarks had a string constant prefix or suffix, the type of expression targeted by Nielsen et al. [44]. Hence, the data show that their technique would likely have at most a small impact on recall for our benchmarks.

A deeper study of inter-procedural property name value flow could provide further insights on how these names are computed; this remains as future work. Still, our data show it is likely that a variety of challenges would need to be addressed to significantly improve ACG's recall with respect to dynamic property accesses.

7.4 Threats to Validity

As noted in Section 6, we do not claim generalizability of the results for our benchmarks to a broader set of JavaScript applications. In our benchmark suite, each individual framework is primarily exercised by a single TodoMVC benchmark, which may not be representative of other applications using that framework. Also, though our harness achieves high statement coverage for the TodoMVC benchmarks (Section 6.3), it is possible that certain application behaviors in those apps remain unexercised. Our dynamic coverage of Juice Shop was relatively low due to scalability limitations; more complete coverage is required to make strong conclusions about relative importance of root causes for that application. Finally, as noted in Section 5, our tooling still does not handle certain language features completely, which may have impacted our measurements.

8 Related Work

Here, we briefly discuss related studies of analysis effectiveness, and also other analysis frameworks and their applicability to framework-based web applications.

Root cause analysis. Our work was partly inspired by a study of call graph recall for Java programs by Sui et al. [57]. As in that work, we measure recall with respect to dynamic analysis measurements, and we aim to determine which constructs are responsible for missing edges. Sui et al.'s approach used calling-context trees [18] and runtime tagging of reflective operations to determine language features impacting recall. Since functions are first-class values in JavaScript, we can trace function data flow directly to make this determination. Also, due to JavaScript's dynamic nature, the potential causes of missing edges and their usage patterns differ significantly from Java's problematic constructs.

Andreasen et al. present techniques for isolating soundness and precision issues in the TAJS static analyzer for JavaScript [20]. For finding analysis unsoundness, their technique creates logs of expression values while executing target programs, and then checks that the static analysis abstractions account for all such values. When unsoundness is discovered for a program, delta debugging [61] is employed to find a reduced version of the program with the same unsoundness. From this reduced program, determining a root cause is often much simpler. In contrast to their work, which is focused on an analysis that strives for full soundness, our approach is targeted at analyses with deliberate unsoundness (for practicality), and aims to quantify the impact of different unsoundness root causes.

Reif et al. [61] present a system that provides methods for exposing sources of unsoundness in different Java call graph builders and also for measuring how frequently hard-to-analyze constructs appear in a set of benchmarks, yielding many useful practical insights. A difference with our work is that our technique can automatically connect specific uses of hard-to-analyze constructs to the corresponding missed call graph edges. This provides important additional information for JavaScript, since hard-to-analyze constructs can appear pervasively in JavaScript code, and not all occurrences cause call graph unsoundness.

Lhoták [37] also presents a comparison of static and dynamic call graphs for Java, aimed at finding sources of imprecision in the static call graph. Other work [20, 60] used dynamic analysis to generate traces and find root causes of imprecision in JavaScript static analyses, and Wei et al. [60] also provides suggestions to fix the root causes of imprecision. Lee et al. [35] produce a tracing graph by tracking information flow from imprecise program points backwards, thereby aiding the user to identify main causes of the imprecision. Our work differs from all of these studies in its focus on recall rather than precision, which necessitates different techniques.

JavaScript Analyses. Several analysis frameworks use abstract interpretation [24] to handle the interdependent problem of scalability and precision in JavaScript [32, 33, 36]. These frameworks have been steadily enhanced with techniques to improve precision and scalability when analyzing libraries, particularly TAJS [19, 31, 32, 43] and SAFE [34, 35, 36, 46, 47, 50]. While these techniques have shown enormous improvement in analyzing libraries like jQuery [10] and Lodash [11], they do not yet scale to complex MVC frameworks like React [4].

Other techniques use dynamic information to improve static analysis. Wei and Ryder introduced blended analysis [59], which uses dynamic analysis to aid static analysis in handling JavaScript's dynamic features. The dynamic flow analysis by Naus and Thiemann [41] generates flow constraints from a training run to infer types in JavaScript applications.

(Their technique finds constraints by tracking operations on values; we determine how values are copied through memory, an orthogonal problem.) Lacuna [45] utilizes static and dynamic analysis to detect dead code in JavaScript applications; this work uses ACG and also uses TodoMVC applications for evaluation. While dynamic information can be very helpful in static analysis, improving pure static analysis is still desirable, as it can compute results without instrumenting and running the code and without inputs.

To analyze JavaScript applications that use the Windows runtime and other libraries, Madsen et al. proposed a use analysis that infers points-to specifications automatically [38]. It is unclear if their analysis will be effective for framework-based applications, where control flow is mainly driven by the framework, not the application. Also, we study applications using diverse frameworks from by many different developers, whereas [38] focuses on Windows libraries. For Node.js, Madsen et al. [39] presented a static analysis using call graphs augmented to represent event-driven control flow. To scale static analysis in server-side JavaScript applications in Node.js, Nielsen et al. present a feedback-driven static analysis to automatically identify the third-party modules that need to be analyzed [42]. Our focus, however, is on client-side MVC applications that often do not have clean module interfaces.

Other recent systems make use of pragmatic JavaScript static analyzers. The CodeQL system [7] includes an under-approximate call graph builder for JavaScript [8]. CodeQL's analysis is primarily intra-procedural, targeted toward taint analysis, and does not handle dynamic property accesses.[10] Møller et al. [40] describe a system for detecting breaking library changes in Node.js programs, based on an under-approximate analysis designed for high recall at the cost of some precision. Nielsen et al. [44] present a pragmatic modular call-graph construction technique for Node.js programs; we discussed its specialized handling of property name expressions in Section 7.3. For these approaches, our methodology could be used to quantify the importance of different causes of reduced recall. Salis et al. recently presented a pragmatic call graph builder for Python programs [51]; it would be interesting future work to extend our techniques to Python. Beyond dataflow-based reasoning about call graphs, other approaches to JavaScript static analysis include AST-based linting [9] and type inference [16, 23].

9 Conclusions

We have presented novel techniques for quantifying the relative importance of different root causes of missed edges in JavaScript static call graphs. We instantiated our approach to perform a detailed study of the results of the ACG algorithm on modern, framework-based web applications. The study's results provided numerous insights on the variety and relative impact of root causes for missed edges. All of our code and data is publicly available. In future work, we plan to extend the study to other domains; we expect that analyses for any dynamic language with extensive use of higher-order functions could benefit from our techniques. We also plan to use the techniques to further develop improved call graph builders and other JavaScript static analyses.

References

1 MDN Web Docs: Object.getOwnPropertyDescriptor(). `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor`, 2021. Accessed: 2021-01-11.

[10] These details are based on personal communication with Max Schäfer in January 2021.

2 MDN Web Docs: with. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with, 2021. Accessed: 2021-01-11.

3 OWASP Juice Shop. https://owasp.org/www-project-juice-shop/, 2021. Accessed: 2021-12-01.

4 React – a JavaScript library for building user interfaces. https://reactjs.org, 2021. Accessed: 2021-01-11.

5 Rhino: JavaScript in Java. https://github.com/mozilla/rhino, 2021. Accessed: 2021-01-11.

6 Angular. https://angular.io, 2022. Accessed: 2022-05-13.

7 CodeQL for research. https://securitylab.github.com/tools/codeql/, 2022. Accessed: 2022-05-13.

8 CodeQL library for JavaScript: Call graph. https://codeql.github.com/docs/codeql-language-guides/codeql-library-for-javascript/#call-graph, 2022. Accessed: 2022-05-13.

9 ESLint. https://eslint.org, 2022. Accessed: 2022-02-25.

10 jquery. https://jquery.com/, 2022. Accessed: 2022-05-13.

11 Lodash. https://lodash.com/, 2022. Accessed: 2022-05-13.

12 MDN Web Docs: Defining Getters and Setters. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_getters_and_setters, 2022. Accessed: 2022-05-13.

13 MDN Web Docs: Function. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function, 2022. Accessed: 2022-05-13.

14 MDN Web Docs: Object.defineProperty(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty, 2022. Accessed: 2022-05-13.

15 Puppeteer. https://pptr.dev/, 2022. Accessed: 2022-05-13.

16 Tern: Intelligent JavaScript Tooling. https://ternjs.net, 2022. Accessed: 2022-02-25.

17 TodoMVC. https://todomvc.com/, 2022. Accessed: 2022-05-13.

18 Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, pages 85–96, 1997.

19 Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, part of SPLASH*, OOPSLA, pages 17–31, 2014.

20 Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the International Workshop on State Of the Art in Program Analysis*, SOAP, pages 31–36, 2017.

21 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Artifact for "Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs", May 2022. doi:10.5281/zenodo.6541325.

22 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs (Extended Version). *arXiv*, 2022. arXiv:2205.06780.

23 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. Type inference for static compilation of JavaScript. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016.

24 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, POPL, pages 238–252, 1977.

25 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *International Conference on Software Engineering*, ICSE, pages 752–761, 2013.

26 Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, 2011.

27 Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *29th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

28 Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 254–263, 2001.

29 Zoltán Herczeg and Gábor Lóki. Evaluation and comparison of dynamic call graph generators for JavaScript. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019*, pages 472–479, 2019.

30 Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis*, ISSTA, pages 34–44, 2012.

31 Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 59–69, 2011.

32 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis, 16th International Symposium*, SAS, pages 238–255, 2009.

33 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for JavaScript. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE, pages 121–132, 2014.

34 Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for JavaScript object-manipulating programs. *Softw. Pract. Exp.*, 49(5):840–884, 2019.

35 Hongki Lee, Changhee Park, and Sukyoung Ryu. Automatically tracing imprecision causes in JavaScript static analysis. *Art Sci. Eng. Program.*, 4(2), 2020.

36 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *In Proceedings of the International Workshop on Foundations of Object Oriented Languages*, FOOL, 2012.

37 Ondrej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE, pages 37–42, 2007.

38 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 499–509, 2013.

39 Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 50(10):505–519, 2015.

40 Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in JavaScript programs affected by breaking library changes. *Proc. ACM Program. Lang.*, 4(OOPSLA):187:1–187:25, 2020. doi:10.1145/3428255.

41 Nico Naus and Peter Thiemann. Dynamic flow analysis for JavaScript. In *Trends in Functional Programming - 17th International Conference*, TFP, pages 75–93, 2016.

42 Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of Node.js applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 455–465, 2019.

43 Benjamin Barslev Nielsen and Anders Møller. Value partitioning: A lightweight approach to relational static analysis for JavaScript. In *34th European Conference on Object-Oriented Programming*, ECOOP, pages 16:1–16:28, 2020.

44 Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of Node.js applications. In Cristian Cadar and Xiangyu Zhang, editors, *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 29–41, 2021. doi:10.1145/3460319.3464836.

45 Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. An extensible approach for taming the challenges of JavaScript dead code elimination. In *25th International Conference on Software Analysis, Evolution and Reengineering*, SANER, pages 391–401, 2018.

46 Changhee Park, Hongki Lee, and Sukyoung Ryu. All about the with statement in JavaScript: removing with statements in JavaScript applications. In *Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH*, DLS, pages 73–84, 2013.

47 Changhee Park, Hongki Lee, and Sukyoung Ryu. Static analysis of JavaScript libraries in a scalable and precise way using loop sensitivity. *Softw. Pract. Exp.*, 48(4):911–944, 2018.

48 Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *Object-Oriented Programming - 25th European Conference*, ECOOP, pages 52–78, 2011.

49 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 1–12, 2010.

50 Sukyoung Ryu, Jihyeok Park, and Joonyoung Park. Toward analysis and bug finding in JavaScript web applications in the wild. *IEEE Softw.*, 36(3):74–82, 2019.

51 Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021.

52 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 165–174, 2013.

53 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 488–498. ACM, 2013.

54 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In David Clarke, Tobias Wrigstad, and James Noble, editors, *Aliasing in Object-Oriented Programming*. Springer, 2013. doi:10.1007/978-3-642-36946-9_8.

55 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Object-Oriented Programming - 26th European Conference*, ECOOP, pages 435–458, 2012.

56 Stack Overflow 2020 Developer Survey: Web Frameworks. `https://insights.stackoverflow.com/survey/2020#technology-web-frameworks`, 2020. Accessed: 2022-05-13.

57 Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *International Conference on Software Engineering*, ICSE, pages 1049–1060, 2020.

58 T.J. Watson Libraries for Analysis (WALA). URL: `http://wala.sourceforge.net`.

59 Shiyi Wei and Barbara G. Ryder. A practical blended analysis for dynamic features in JavaScript. Technical Report TR-12-18, Virginia Tech, 2012. URL: `https://vtechworks.lib.vt.edu/handle/10919/19421`.

60 Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 487–498, 2016.

61 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types

Nicolas Lagaillardie
Department of Computing, Imperial College London, UK

Rumyana Neykova
Department of Computer Science, Brunel University London, UK

Nobuko Yoshida
Department of Computing, Imperial College London, UK

Abstract

Communicating systems comprise diverse software components across networks. To ensure their robustness, modern programming languages such as Rust provide both strongly typed channels, whose usage is guaranteed to be *affine* (*at most once*), and *cancellation* operations over *binary* channels. For coordinating components to correctly communicate and synchronise with each other, we use the structuring mechanism from *multiparty session types*, extending it with affine communication channels and implicit/explicit cancellation mechanisms. This new typing discipline, *affine multiparty session types* (AMPST), ensures *cancellation termination* of multiple, independently running components and guarantees that communication will not get stuck due to error or abrupt termination. Guided by AMPST, we implemented an automated generation tool (`MultiCrusty`) of Rust APIs associated with cancellation termination algorithms, by which the Rust compiler auto-detects unsafe programs. Our evaluation shows that `MultiCrusty` provides an efficient mechanism for communication, synchronisation and propagation of the notifications of cancellation for arbitrary processes. We have implemented several usecases, including popular application protocols (OAuth, SMTP), and protocols with exception handling patterns (circuit breaker, distributed logging).

2012 ACM Subject Classification Software and its engineering → Software usability; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi

Keywords and phrases Rust language, affine multiparty session types, failures, cancellation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.4

Related Version *Full Version*: `https://arxiv.org/abs/2204.13464`

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.9`

Funding The work is supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T014709/1 and EP/V000462/1, and NCSS/EPSRC VeTSS.

Acknowledgements We thank the ECOOP reviewers for their insightful comments and suggestions, and (alphabetical order) Zak Cutner, Wen Kokke, Roland Kuhn, Dimitris Mostrous and Martin Vassor for discussions.

1 Introduction

The advantage of message-passing concurrency is well-understood: it allows cheap horizontal scalability at a time when technology providers have to adapt and scale their tools and applications to various devices and platforms. In recent years, the software industry has seen a shift towards languages with native message-passing primitives (e.g., Go, Elixir and Rust). Rust, for example, has been named the most loved programming language in the

© Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida;
licensed under Creative Commons License CC-BY 4.0

annual Stack Overflow survey for five consecutive years (2016-20) [49]. It has been used for the implementation of large-scale concurrent applications such as the Firefox browser, and Rust libraries are part of the Windows Runtime and Linux kernel. Rust's rise in popularity is due to its efficiency and memory safety. Rust's dedication to safety, however, does not *yet* extend to communication safety. Message-passing based software is as liable to errors as other concurrent programming techniques [50] and communication programming with Rust built-in message-passing abstractions can lead to a plethora of communication errors [30].

Much academic research has been done to develop rigorous theoretical frameworks for the verification of message-passing programs. One such framework is *multiparty session types* (MPST) [21] – a type-based discipline that ensures concurrent and distributed systems are *safe by design*. It guarantees that processes following a predefined communication protocol (also called a *multiparty session*) are free from communication errors and deadlocks. Rust may seem a particularly appealing language for the practical embedding of session types with its message-passing abstractions and affine type system. The core theory of session types, however, has serious shortcomings as its safety is guaranteed under the assumption that a session should run until its completion without any failure. Adapting MPST in the presence of failure and realising it in Rust are closely intertwined, and raise two major challenges:

Challenge 1: Affine multiparty session types (AMPST). There is an inherent conflict between the affinity of Rust and the linearity of session types. The type system of MPST guarantees a *linear* usage of channels, i.e., communication channels in a session must be used *exactly once*. As noted in [30], in a distributed system, it is a common behaviour that a channel or the whole session can be cancelled *prematurely* – for example, a node can suddenly get disconnected, and the channels associated with that node will be dropped. A naive implementation of MPST cancellation, however, will lead to incorrect error notification propagation, orphan messages, and stuck processes. The current theory of MPST does not capture affinity, hence cannot guarantee deadlock-freedom and liveness between multiple components in a realistic distributed system. Classic multiparty session type systems [21] do not prevent any errors related to session cancellation. An affine multiparty session type system should (1) prevent infinitely cascading errors, and (2) ensure deadlock-freedom and liveness in the presence of session cancellations for arbitrary processes. Although there are a few works on affine session types, they are either binary [39, 16] or modelling a very limited cancellation over a single communication action, and a general cancellation is not supported [19] (see § 6.2, and [33]).

Challenge 2: Realising an affine multiparty channel over binary channels. The extension of binary session types to multiparty is usually not trivial. The theory assumes multiparty channels, while channels, in practice, are binary. To preserve the global order specified by a global protocol, also called the order of interactions, when implementing a multiparty protocol over binary channels, existing works [22, 40, 44, 6] use code generation from a global protocol to local APIs, requiring type-casts *at runtime* on the underlying channels, compromising the type safety of the host type system. Implementing MPST with failure becomes especially challenging given that cancellation messages should be correctly propagated across multiple binary channels.

In this work, we overcome the above two challenges by presenting a new affine multiparty session types framework for Rust (AMPST). We present a shallow embedding of the theory into Rust by developing a library for safe communication, `MultiCrusty`. The library utilises a new communication data structure, *affine meshed channels*, which stores multiple binary

channels without compromising type safety. A macro operation for exception handling safely propagates failure across all in-scope channels. We leverage an existing binary session types library, Rust's macros facilities, and optional types to ensure that communication programs written with `MultiCrusty` are *correct-by-construction*.

Our implementation brings three insightful contributions: (1) multiparty communication safety can be realised by the native Rust type system (without external validation tools); (2) top-down and bottom-up approaches can co-exist; (3) Rust's destructor mechanism can be utilised to propagate session cancellation. All other works generate not only the types but also the communication primitives for multiparty channels which are protocol-specific. The crucial idea underpinning the novelty of our implementation is that one can pre-generate the possible communication actions without having the global protocol; and then use the types to limit the set of permitted actions. Without this realisation neither (1), nor (2) is possible.

Paper Summary and Contributions

§ 2 outlines the gains of programming with *affine meshed channels* by introducing our running example, a Video streaming service, and its Rust implementation using `MultiCrusty`.

§ 3 establishes the metatheory of AMPST. We present a core multiparty session π-calculus with session delegation and recursion, together with new constructs for exception handling, and affine selection and branching (from Rust optional types). The calculus enjoys session-fidelity (Theorem 3.14), deadlock-freedom (Theorem 3.16), liveness (Theorem 3.17), and a novel cancellation termination property (Theorem 3.22).

§ 4 describes the main challenges of embedding AMPST in Rust, and the design and implementation of `MultiCrusty`, a library for safe multiparty communication Rust programming.

§ 5 evaluates the execution and compilation overhead of `MultiCrusty`. Microbenchmarks show negligible overhead when compared with the built-in unsafe Rust channels, provided by `crossbeam-channel`, and up to two-fold runtime improvement to a binary session types library on protocols with high-degree of synchronisation. We have implemented, using `MultiCrusty`, examples from the literature, and application protocols (see [33]).

Additionally, § 6 discusses related works and concludes. The proofs of our theorems are included in [33]. Our library is available in this public library: `https://github.com/NicolasLagaillardie/mpst_rust_github/`. An ECOOP artifact is also available.

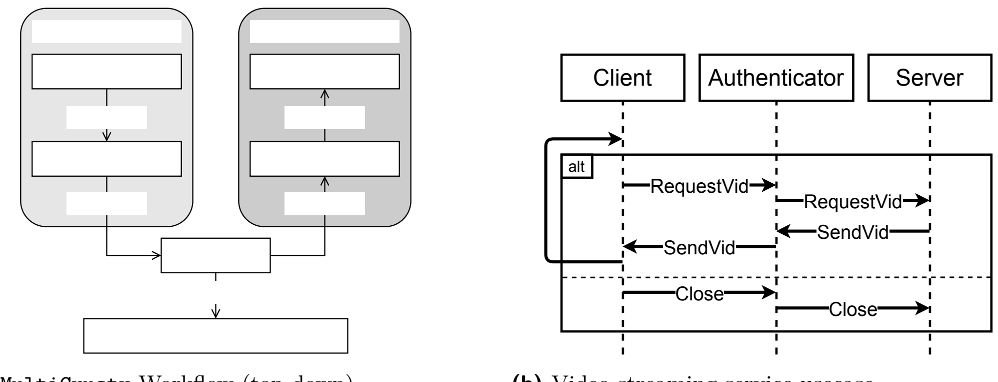
2 Overview: affine multiparty session types (AMPST) in Rust

Framework overview: AMPST in Rust

Figure 1a depicts the overall design of `MultiCrusty`. Our design combines the top-down [21] and bottom-up [34] methodologies of multiparty session types in a single framework. Our bottom-up approach is discussed in details in [33].

The top-down approach enables *correctness-by-construction* and requires that a developer specifies a global type (hereafter a global protocol) describing the communication behaviour of the program. We utilise the Scribble toolchain [47] for writing and verifying global protocols. The toolchain projects local types for each role in a protocol. We have augmented the toolchain to further generate those local types into Rust types, i.e., types that stipulate the behaviour of communication channels.

Our Rust API (`MultiCrusty` API) integrates both approaches, as illustrated in Figure 1a. Developers can choose to either (1) write the global protocol and have the Rust types generated, or (2) write the Rust types manually and check that the types are compatible.



(a) `MultiCrusty` Workflow (top-down). **(b)** Video streaming service usecase.

Figure 1 Programming with multiparty session types.

```
1  // generates at compile-time communication primitives for 3-party affine meshed channels
2  gen_mpst!(MeshedChannelsThree, A, C, S);
```

```
1  fn client(
2   s: RecC<i32>,
3   i: i32
4   ) -> R {
5   if (i<MAX) {
6   let s = choose_c!(s,
7   ChoiceA::Video, ChoiceS::
        Video)
8   let n = get_video(i);
9   let s = s.send(n)?;
10  let (_,s) = s.recv()?;
11  client(s, i+1)
12  } else {
13  let s = choose_c!(s,
14  ChoiceA::Close, ChoiceS::
        Close);
15  s.close()
16  }
17 }
```

```
1  fn auth(s: RecA<i32>)
2   -> R {
3   offer_mpst!(
4   s, {
5   ChoiceA::Video(s)
6   => {
7   let (x,s) = s.recv()?;
8   let s = s.send(x)?;
9   let (x,s) = s.recv()?;
10  let s = s.send(x)?;
11  auth(s)
12  },
13  ChoiceA::Close(s)
14  => {
15  s.close()
16  } }
17  )
18 }
```

```
1  fn server(s: RecS<i32>)
2   -> R {
3   offer_mpst!(
4   s, {
5   ChoiceS::Video(s)
6   => {
7   let v = attempt!{{
8    let (x, s) = s.recv()?;
9    let f = get_file(x);
10   read_video_file(f)
11  } catch (e) {
12   cancel(s);
13   panic!("Err: {:?}", e)
14  } }()?;
15  let s = s.send(x)?;
16  server(s)}
17  ChoiceS::Close(s)
18  => {s.close()
19 } } ) }
```

(a) role C. **(b)** role A. **(c)** role S.

a!*ReqVideo* a?*ResVideo* a!*close*

c?*ReqVideo* s!*ReqVideo* c!*ResVideo* s?*ResVideo* c?*close* s!*close*

a?*ReqVideo* a!*ResVideo* a?*close*

Figure 2 Rust implementations and respective CFSMs of role C (a), role A (b) and role S (c).

Note that both approaches rely on concurrent programs written with `MultiCrusty` API, and both approaches rely on the Rust compiler to type check the concurrent programs against their respective types. Overall, the framework guarantees that well-typed concurrent programs implemented using `MultiCrusty` API with Scribble-generated types or k-MC-compatible types, will be free from deadlocks, reception errors, and protocol deviations.

The main primitives of `MultiCrusty` API are summarised in Table 1. Next, we briefly explain them through an example. A more detailed explanation is provided in § 4.

2.1 Example: Video streaming service

The *Video streaming service* is a usecase that can take full advantage of affine multiparty session types and demonstrate the need for multiparty channels with cancellation. Each streaming application connects to servers, and possibly other devices, to access services and follows a specific protocol. To present our design, we use a simplified version of the protocol, omitting the authentication part, illustrated in the diagram of Figure 1b. The diagram should be read from top to bottom. The protocol involves three services – an *Authenticator* (role $\boldsymbol{A}$) service, a *Server* (role $\boldsymbol{S}$) and a *Client* (role $\boldsymbol{C}$). The protocol starts with a choice on the *Client* to either request a video or end the session. The first branch is, *a priori*, the main service provided, i.e., request for a video. The *Client* cannot directly request videos from the *Server* and has to go through the *Authenticator* instead. On the diagram, the choice is denoted as the frame `alt` and the choices are separated by the horizontal dotted line. The protocol is recursive, and the *Client* can request new videos as many times as needed. This recursive behaviour is represented by the arrow going back on the *Client* side in Figure 1b. To end the session, the *Client* first sends a `Close` message to the *Authenticator*, which then subsequently sends a `Close` message to the *Server*.

Affine meshed channels and multiparty session programming with `MultiCrusty`

The implementations in `MultiCrusty` of the three roles are given in Figure 2. They closely follow the behaviour that is prescribed by the protocol. The global protocol does not explicitly specify cancellation. However, in a distributed setting, timeout or failure can happen at any time: a request from a CDN network or cloud storage to the server might be a timeout or the result message might be lost. Our implementation accounts for failure by providing communication primitives for two different types of session cancellation, called *implicit* and *explicit*: either we run a block of code and upon any error at any point, we go to the **catch** branch, or we explicitly test each step.

The implementation of the concurrent programs starts by generating at compile-time all communication primitives for affine channels between the roles. This is done by the macro `gen_mpst!(MeshedChannelsThree, A, C, S)`, see line 2 in Figure 2. The macro `gen_mpst!` takes two kinds of arguments: a string literal which represents the name of the data structure for affine meshed channels (`MeshedChannelsThree` here), and a string literal for each role (`A`, `C` and `S` here, can be any number of roles).

To explain affine meshed channels and all `MultiCrusty` communication primitives, we focus on the implementation for role $\boldsymbol{A}$ given in Figure 2b. The implementations of the other roles are similar. First, line 1 declares an `auth(s)` function that is parametric on an affine meshed channel `s` of type `RecA<i32>`, the result type of the function is irrelevant to our explanation, hence we have simply denoted it by `R`. The type `RecA<i32>`, an alias for the full type described in Figure 6, specifies the operations allowed on `s`. As mentioned previously, this type can be either written by the developer or generated by Scribble. We defer the explanation of the (generated) types to § 4, i.e., the full Rust type is given in Figure 6. For clarity, here we only give a high-level view of the behaviour for each channel by representing its respective local session types as a communicating finite state machine (CFSM [4]), where ! (resp. ?) denotes sending (resp. receiving). The CFSMs for each role (channel) can be seen in Figure 2. For example, `c!ResVideo` means that role $\boldsymbol{A}$ is receiving from the role $\boldsymbol{C}$ a message labelled as `Video`, while `s!ReqVideo` says that role $\boldsymbol{A}$ sends a message to role $\boldsymbol{S}$.

Table 1 Primitives provided by `MultiCrusty`. `s` is an affine meshed channel; `p` is a payload of a given type; I is a subset of all roles in the protocol but the current role; K is a subset of all branches.

Primitives	Description
`let s = s.send(p)?;`	Sends a payload `p` on a channel `s` and assigns the continuation of the session (a new meshed channel) to a new variable `s`.
`let (p, s) = s.recv()?;`	Receives a payload `p` on channel `s` and assigns the continuation of the session to a new variable `s`.
`s.close()`	Closes the channel `s` and returns a unit type.
`offer_mpst!(s, {` $\texttt{enum}_\texttt{i} :: \texttt{variant}_\texttt{k}\texttt{(e)} \Rightarrow \{...\}_{k \in K}$ `})`	Role i receives a message label on channel `s`, and, depending on the label value which should match one of the variants $\texttt{variant}_\texttt{k}$ of $\texttt{enum}_\texttt{i}$, runs the related block of code.
`choose!(s,` $\{\texttt{enum}_\texttt{i} :: \texttt{variant}_\texttt{k}\}_{i \in I}$ `)`	Sends the chosen label, which corresponds to $\texttt{variant}_k$, to all other roles.
`attempt! {{ ... } catch(e) { ... }}`	Attempts to run the first block of code and, upon error, catches the error in the variable `e` and runs the second block of code.

The thread for role ***A*** uses an affine meshed channel `s` to implement the given CFSM behaviour. In essence, the meshed channel is implemented as an indexed tuple of binary channels – one binary channel for each pair of interacting processes, i.e., a binary channel for role ***A*** and role ***S*** and a binary channel for role ***A*** and role ***C***.

The implementation starts by realising a choice: role ***C*** broadcasts its choice, which can either be to request a video at line 7 or to close the connection at line 14 (Figure 2a). Role ***C*** broadcasts the choice to every other role. This choice is received by role ***A***, which will either receive a `Video` or a `Close` label. This behaviour is implemented by the `MultiCrusty` macro `offer_mpst!` (line 3), which is applied to a multiparty channel `s` and a sum type, either `ChoiceA::Video` or `ChoiceA::Close` here. The behaviour of each branch in the protocol is implemented as an anonymous function.

Lines 5 – 12 supply such a function that implements the behaviour when role ***C*** requests a `Video`, while lines 13 – 16 handle the `Close` request. At each step, only one of the primitives available in Table 1 (but the `attempt!` macro) is available, linked to the type of channel `s`. Those primitives will either return the expected continuation as well as additional variables if necessary, such as the payload for `recv()`, or an error if the operation failed.

The types of the affine meshed channels, as well as the generic types in the declaration of the `MultiCrusty` communication functions, enable compile-time detection of protocol violations. Examples of protocol violations include swapping lines 10 and 9, using another communication primitive or using the wrong payload type. The Rust type system, on the other hand, ensures that all affine channels are used at most once. For example, using channel `s` twice (without rebinding) will be detected by the compiler. All the errors mentioned above will be reported as compile-time errors. In the case that an unexpected runtime error occurs, all roles are guaranteed to terminate safely. This is ensured by two mechanisms – explicit session cancellation (that can be triggered by the user) and implicit session cancellation (that is embedded in the `MultiCrusty` primitives and the channel destructors).

Implicit and explicit cancellations

The processes of role ***A*** and role ***S*** illustrate resp. implicit and explicit cancellations. The primitive `cancel(s)` drops the affine meshed channel `s`, and its binary channels, making it inaccessible to other participants. This is convenient when an error related to the computation aspects of the program occurs. For example, in Figure 2c the session is cancelled in line 12 after an error occurs as a result of reading a corrupted video. We used the `attempt!{ { ... } catch(e) { ... } }` macro (Rust version of a `try-catch` block) in lines 7 to 14 to

catch the error message, and explicitly cancel the session. The macro tries to go through the `attempt`-block of code, and upon any error in this block, stops the process and calls the `catch(e)`-block with the error message `e`. Line 13 executes a `panic!`, which allows a program to terminate immediately and provide feedback to the caller of the program. Calling `cancel(s)` then `panic!` or calling `panic!` alone will result in the same outcome. Forgetting both will throw an error because the output type will not match the one of `fn server(s)`, unless replaced with an `Ok(())`. In any cases, an error will be thrown on other threads linked to other roles because role S's sessions are inaccessible in the `catch(e)`-block.

Alternatively, we explain implicit cancellation as implemented by role A in Figure 2b. The construct `let x = f()?`, as seen in line 7, is Rust's *monadic bind* notation for programs and functions that may return errors: their usual output type is `Result<T, Error>` where `T` is the expected type if everything goes right and `Error` is the error type returned. For any program and function returning such type, the users have to *unwrap* it. The two usual ways of doing so are by using the `?`-operator, or by pattern matching on the result using `match`. In our case, if `recv()` succeeds, the `?`-operator unpacks the result and returns the tuple containing the received payload and the continuation. If `recv()` fails, the `?`-operator short-circuits, skips the rest of the statements, and returns the error. We use this mechanism to catch any session cancellation. In the case that a `recv()` (or `send()`) does not succeed, the implementation of the underlying communication primitive will cancel the channel and broadcast the cancellation to all other binary channels that are part of the session.

Finally, we look at the implementation of role C to demonstrate the final mechanism of session cancellation. For this purpose, we have to comment out lines 9 – 11 in Figure 2a or replace them with a `panic!` as to simulate a wrongly implemented role C. With such modification, this function will still compile despite the protocol not being fully implemented (since the last received action from role A is missing, the meshed channel `s` will be dropped prematurely). Even in this case, `MultiCrusty` ensures that all processes will terminate safely, i.e., all parties are notified that an affine channel has been dropped. Prematurely dropping a channel can happen due to incorrectly implemented behaviour (as we demonstrated above), or by unhandled user error, for example, the function `get_video()` in line 8 can invoke a `panic!` because there is no video associated with the index `i`. Safe session termination is realised by customising the native destructor `Drop` in Rust, as proposed for binary meshed channels by [30]. When an affine meshed channel goes out of scope, the channel destructor is called, the session is cancelled, dropping every channel value used in the session, and only then is the memory deallocated. Deallocations, whether they come from a cancellation or a channel closure, lead to the safe collection of the variables by the stable Rust compiler we rely on, avoiding memory leaks.

In short, a session can be cancelled for three reasons: (1) an error affecting the computation aspects of the program, as in Figure 2c; (2) an error during communication, e.g., a timeout on a channel, as in Figure 2b; or (3) a premature drop of the affine meshed channel due to incorrect implementation, as in Figure 2a. Our mechanisms for session cancellation cover all the above cases. In this way, our framework provides *affine multiparty session compliance* by ensuring that (1) if all results are returned without failure, the processes follow the given Scribble global protocol (Theorem 3.14) or (2) once a cancellation happens, all processes in the same session terminate with an error (Theorem 3.22). We have proven the above results by formalising affine meshed channels in an extension of a multiparty π-calculus.

3 Affine multiparty session processes for Rust programming

3.1 Affine multiparty session processes

Our calculus (AMPST) is an extension of a full multiparty session π-calculus [46] which includes session delegation (channel passing) and session recursion. We shade additions to [46] in this colour .

▶ **Definition 3.1.** The **affine multiparty session π-calculus** (AMPST) is defined as follows:

$$
\begin{array}{rcll}
c, d & ::= & x \mid s[\mathtt{p}] \qquad \dagger \;::=\; \emptyset \mid \textbf{?} & \text{(variable, channel with role } \mathtt{p}\text{, error, flag)} \\
P, Q & ::= & \mathbf{0} \mid P \mid Q \mid (\nu s)\, P & \text{(inaction, composition, restriction)} \\
 & & \textbf{?}\, c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P \mid \textbf{?}\, c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i & \text{(affine selection, branching } I \neq \emptyset\text{)} \\
 & & c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P \mid c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i & \text{(selection, branching } I \neq \emptyset\text{)} \\
 & & \textbf{def}\ D\ \textbf{in}\ P \mid X\langle\tilde{c}\rangle & \text{(process definition, process call)} \\
 & & \textbf{try}\ P\ \textbf{catch}\ Q \mid \textbf{cancel}(c).P \mid s\lightning & \text{(catch, cancel, kill)} \\
D & ::= & X(\tilde{x}) = P & \text{(declaration of process variable } X\text{)}
\end{array}
$$

A set $\mathscr{P}$ denotes **participants**: $\mathscr{P} = \{\mathtt{p}, \mathtt{q}, \mathtt{r}, \dots\}$, and $\mathbb{A}$ is a set of alphabets. A **channel** c can be either a variable or a **channel with role** $s[\mathtt{p}]$, i.e., a multiparty communication endpoint whose user plays role $\mathtt{p}$ in the session s. $\tilde{c}$ denotes a vector $c_1 c_2 \dots c_n$ $(n \geq 1)$ and similarly for $\tilde{x}$ and $\tilde{s}$.

The two processes with $\textbf{?}$ model the option `?`-operator in Rust. Process $\textbf{?}\, c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P$ performs an **affine selection (internal choice)** towards role $\mathtt{q}$, using the channel c: if the *message label* $\mathtt{m}$ with the *payload* channel d is successfully sent, then the execution continues as P; otherwise (if the receiver has failed or timeout), it triggers an exception. The **affine branching (external choice)** $\textbf{?}\, c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i$ uses channel c to wait for a message from role $\mathtt{q}$: if a message label $\mathtt{m}_k$ with payload d is received (for some $k \in I$), then the execution continues as P_k, with x_k replaced by d; if not received, it triggers an exception. Note that message labels $\mathtt{m}_i$ are pairwise distinct and their order is irrelevant, and variable x_i is bound with scope P_i.

The following two failure handling processes follow the program behaviour of Figure 2c. The **try-catch** process, **try** P **catch** Q, consists of a *try process* P which is ready to communicate with parallel composed one; and a *catch process* Q which becomes active when a cancellation or an error happens. The **cancel** process, $\textbf{cancel}(c).P$, cancels other processes whose communication channel is c. The **kill** $s\lightning$ kills all processes with session s and **is generated only at runtime** from affine or cancel processes.

The other syntax is from [46]. The **inaction 0** represents a terminated process (and is often omitted). The **parallel composition** $P \mid Q$ represents two processes that can execute concurrently, and potentially communicate. The **session restriction** $(\nu s)\, P$ declares a new session s with a scope limited to process P. The linear **selection** and the linear **branching** can be understood as their affine versions but without failure handling. **Process definition**, **def** $X(\tilde{x}) = P$ **in** Q and **process call** $X\langle\tilde{c}\rangle$ model recursion: the call invokes X by expanding it into P, and replacing its formal parameters with the actual ones.

Linear or affine branching and selection are denoted as either $\dagger\, c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i$ and $\dagger\, c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P$. We use $\mathrm{fv}(P)\,/\,\mathrm{fc}(P)$ and $\mathrm{dpv}(P)\,/\,\mathrm{fpv}(P)$ to denote *free variables/channels* and *bound/free process variables* of P. We call a process P such that $\mathrm{fv}(P) = \mathrm{fpv}(P) = \emptyset$ *closed*. A *set of subjects of* P, written $\mathrm{sbj}(P)$, is defined as: $\mathrm{sbj}(\mathbf{0}) = \emptyset$; $\mathrm{sbj}(P \mid Q) = \mathrm{sbj}(P) \cup \mathrm{sbj}(Q)$; $\mathrm{sbj}((\nu s)\, P) = \mathrm{sbj}(P) \setminus \{s[\mathtt{p}_i]\}_{i\in I}$; $\mathrm{sbj}\big(\dagger\, c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i\big) = \mathrm{sbj}(\dagger\, c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P) = \{c\}$; $\mathrm{sbj}(\textbf{def}\ X(\tilde{x}) = P\ \textbf{in}\ Q) = \mathrm{sbj}(Q) \cup \mathrm{sbj}(P) \setminus \{\tilde{x}\}$ with $\mathrm{sbj}(X\langle\tilde{c}\rangle) = \mathrm{sbj}(P\{\tilde{c}/\tilde{x}\})$; $\mathrm{sbj}(\textbf{try}\ P\ \textbf{catch}\ Q) = \mathrm{sbj}(P)$; and $\mathrm{sbj}(\textbf{cancel}(c).P) = \{c\}$.

[R-Com] $\mathbb{E}_1[\dagger\, s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i] \mid \mathbb{E}_2[\dagger\, s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{m}_k\langle s'[\mathtt{r}]\rangle.Q] \;\rightarrow\; P_k\{^{s'[\mathtt{r}]}/_{x_k}\} \mid Q \quad \text{if } k\in I$

[C-?Sel] $?\, s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P \rightarrow s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P \mid s\lightning$

[T?Sel] $\textbf{try}\ ?\, s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P\ \textbf{catch}\ Q \rightarrow Q \mid s\lightning$

[C-Sel] $s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P \mid s\lightning \rightarrow P \mid s\lightning \mid s'\lightning$

[C-?Br] $?\, s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i \rightarrow s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s\lightning$

[T?Br] $\textbf{try}\ ?\, s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i\ \textbf{catch}\ Q \rightarrow Q \mid s\lightning$

[C-Br] $s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s\lightning \rightarrow (\nu s')\,(P_k\{^{s'[\mathtt{r}]}/_{x_k}\} \mid s'\lightning) \mid s\lightning \quad s'\notin \mathrm{fc}(P_k)\,, k\in I$

[R-Can] $\mathbb{E}[\textbf{cancel}(s[\mathtt{p}]).Q] \rightarrow s\lightning \mid Q$ [C-Cat] $\textbf{try}\ P\ \textbf{catch}\ Q \mid s\lightning \rightarrow Q \mid s\lightning \quad \exists\mathtt{r}.\ s[\mathtt{r}] = \mathrm{sbj}(P)$

[R-Def] $\textbf{def}\ X(x_1,\ldots,x_n) = P\ \textbf{in}\ (X\langle s_1[\mathtt{p}_1],\ldots,s_n[\mathtt{p}_n]\rangle \mid Q)$
$\rightarrow \textbf{def}\ X(x_1,..,x_n) = P\ \textbf{in}\ (P\{^{s_1[\mathtt{p}_1]}/_{x_1}\}\cdot\cdot\{^{s_n[\mathtt{p}_n]}/_{x_n}\} \mid Q)$

[R-Ctx] $P \rightarrow P'$ implies $\mathbb{C}[P] \rightarrow \mathbb{C}[P']$ [R-Struct] $P \equiv P' \rightarrow Q' \equiv Q$ implies $P \rightarrow Q$

Figure 3 AMPST π-calculus reduction between closed processes (we highlight the new rules from [46])

The set of subjects is the key definition which enables us to define the typing system for the `try-catch` process with recursive behaviours.

▶ **Example 3.2** (Subjects of processes). Assume $R_1 = \textbf{def}\ X(x) = x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0}\ \textbf{in}\ X\langle c\rangle$ which repeats the action at c and emits a message d with label repeatedly interacting with the dual input (but reduction with this process only happens if there is a corresponding input at c, i.e., on-demand). We calculate $\mathrm{sbj}(R_1)$ as:

$$\mathrm{sbj}(\textbf{def}\ X(x) = x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0}\ \textbf{in}\ X\langle c\rangle) = \mathrm{sbj}(X\langle c\rangle)\cup\mathrm{sbj}(\textbf{def}\ X\langle x\rangle = x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0})$$
$$= \mathrm{sbj}(X\langle c\rangle) = \mathrm{sbj}((x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0})\{^c/_x\}) = \{c\}$$

Another example is: $\mathrm{sbj}(\textbf{try}\ x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0}\ \textbf{catch}\ \textbf{cancel}(x[\mathtt{q}]).\mathbf{0}) = \mathrm{sbj}(x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0}) = \{x\}$.

▶ Remark 3.3 (Syntax and semantics). AMPST extends MPST incorporating some design choices from [39], aiming to distil the implementation essence of `MultiCrusty`. The design of our `try-catch` process follows the binary affine session types in [39], but models more cancellations for arbitrary processes with affine branchings/selections and cancel processes non-deterministically (whose semantics follow the implementation behaviours, see § 4.4). We list the essential differences from [39].

(1) (Nondeterministic failures) The kill process is a runtime syntax and generated only during reductions unlike [39]. Our calculus also allows *nondeterministic failures* caused by either (1) affine selection/branching or (2) `try-catch` processes. See [33] for examples.

(2) (Recursion parameterised by linear names) One of the novelties of our formalism which is not found in [39] is a combination of session recursions, affinity, and interleaved sessions, i.e., the **def** agents (linearly parameterised recursions), which are the most technical part when designing the typing system with `try-catch` processes. The combination of all features is absent from [39, 16, 19]: see § 6 for more detailed comparisons.

▶ **Definition 3.4** (Semantics). *A* ***try-catch context*** $\mathbb{E}$ *is:* $\mathbb{E} ::= \textbf{try}\ \mathbb{E}\ \textbf{catch}\ P \mid [\,]$ *and a* ***reduction context*** $\mathbb{C}$ *is:* $\mathbb{C} ::= (\nu s)\,\mathbb{C} \mid \textbf{def}\ D\ \textbf{in}\ \mathbb{C} \mid \mathbb{C} \mid P \mid P \mid \mathbb{C} \mid [\,]$. ***Reduction*** $\rightarrow$ *is inductively defined in Figure 3, which uses the* ***structural congruence*** $\equiv$ *which is defined by* $s\lightning \mid s\lightning \equiv s\lightning$ *and* $(\nu s)\,s\lightning \equiv \mathbf{0}$ *together with other rules in [46].*

▶ Remark 3.5 (Nested try-catches and $\mathbb{E}$). The context $\mathbb{E}$ is only used for defining the reductions at the top parallel composed processes, *not* used nested exception handling like [16, 19]. Our (typable) try-catch processes allow any form of processes such as recursions, parallel, session delegations, and restriction/scope opened processes under a guarded process:

$$R = \mathbf{try}\ s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{m}_1.(\nu s')\,(s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{m}_3\langle s'[\mathtt{r}]\rangle.\mathbf{0} \mid \mathbf{try}\ s'[\mathtt{q}][\mathtt{r}]\oplus\mathtt{m}_2.\mathbf{0}\ \mathbf{catch}\ \mathbf{cancel}(s'[\mathtt{q}]).\mathbf{0})\\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0}$$

See [33] for more typable processes with nested `try-catch` blocks.

We explain each rule highlighting the new rules.

Communication. Rule [R-Com] is the main communication rule between an affine/linear selection and an affine/linear branching. Linear selections/branching are placed in the *try* position but can interact with affine counterparts. Once they interact, processes are spawned from *try-blocks* (notice that $\mathbb{E}_1$, $\mathbb{E}_2$ are erased after the communication), and start communicating on parallel with other parallel composed processes. Note that the context $\mathbb{E}$ is discarded after the successful communication.

Error-Cancellation. Rules [C-?Sel] and [C-?Br] model the situations that an error handling occurs at the affine selection/branching. This might be the case if its counterpart has failed (hence [R-Com] does not happen) or timeout. It then triggers the kill process at s. Rules [T?Sel] and [T?Br] model the case that the affine selection/branching are placed inside the try-block and triggered by the error. In this case, it will go to the *catch-block*, generating a kill process.

Cancelling Processes. Rule [C-Sel] cancels the selection prefix s, additionally generating the kill process at the delegated channel for all the session processes at s' to be cancelled. Rule [C-Br] cancels only one of the branches – this is sufficient since all branches contain the same channels except x_i (ensured by rule [T-&] in Figure 4). After the cancellation, it additionally instantiates a fresh name $s'[\mathtt{r}]$ to x_k into P_k. The generated kill process at s' kills prefixes at $s'[\mathtt{r}]$ in $P_k\{^{s'[\mathtt{r}]}/_{x_k}\}$.

Cancellation from Other Parties. Rule [R-Can] is a cancellation and generates a kill process. Note that the `try-catch` context $\mathbb{E}$ is thrown away. Rule [C-Cat] is prompted to move to Q by kill $s\lightning$. The side condition $\mathrm{sbj}(P)$ ensures that P is a prefix at s (up to $\equiv$ for a recursive process). All mimic the behaviour of the programs in Figure 2c.

Other Rules. Rules [R-Def], [R-Ctx], and [R-Struct] are standard from [46]. In Figure 3, the two new rules are for garbage collections of kill processes.

▶ **Example 3.6** (Syntax and reductions). A process might be completed, or cancelled in many ways, and also interacts non-deterministically. We demonstrate the reduction rules using the running example with a minor modification. We use a nested `try-catch` block, and for simplicity we use shorter label names, and we use a constant, i.e., d, as a message payload.

Assume the process for role S is $P = ?\,s[\mathtt{p}][\mathtt{q}](Q + \mathtt{close}(x).\mathbf{0})$ where

$$Q = \mathtt{video}(x).\,\mathbf{try}\ ?\,s[\mathtt{p}][\mathtt{q}]\mathtt{req}(x).\mathbf{try}\ ?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0}\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0}$$

The following shows a possible reduction.

$$P \mid s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{video}\langle \mathtt{d}\rangle.s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{req}\langle \mathtt{d}\rangle.s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \quad (1)$$

$$\text{[R-Com]} \rightarrow \ \mathbf{try}\ (?\,s[\mathtt{p}][\mathtt{q}]\mathtt{req}(x).\mathbf{try}\ (?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle \mathtt{d}\rangle.\mathbf{0})\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0})\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0} \quad (2)$$

$$\mid s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{req}\langle \mathtt{d}\rangle.s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \quad (3)$$

$$\text{[R-Com]} \rightarrow \mathbf{try}\ ?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle \mathtt{d}\rangle.\mathbf{0}\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0} \mid s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \quad (4)$$

$$\text{[T?Sel]} \rightarrow \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0} \mid s\lightning \mid s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \quad (5)$$

$$\text{[R-Can]} \rightarrow s\lightning \mid s\lightning \mid \mathbf{0} \mid s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \quad (6)$$

$$\text{[C-Br]} \rightarrow s\lightning \mid s\lightning \mid \mathbf{0} \mid \mathbf{0} \equiv s\lightning \quad (7)$$

$\mathbb{E}_6[P_6]$ for Equation (1) is $P_6 = ?\,s[\mathtt{p}][\mathtt{q}]\mathtt{req}(x).\mathbf{try}\ \ldots\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0}$ and $\mathbb{E}_9[P_9]$ for Equation (4) is $P_9 = ?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle \mathtt{d}\rangle.\mathbf{0}$ both because of rule [C-Cat].

Initially we reduce using the communication rule for the branching and selection. Next, we apply [R-Com] demonstrating how the affine branching reduces under **try**. Then we apply [T?Sel] assuming an error (or a timeout) occurs during the selection of $\mathtt{res}$. This generates a kill process $s\lightning$ and spawns the process in the **catch**-block. **Cancel** spawns a kill process $s\lightning$ and hence reduces to $s\lightning \mid \mathbf{0}$, following rule [R-Can] with $\mathbb{E} = [\,]$. Finally, applying [R-Can] cancels the linear selection. To conclude, we garbage collect all kill processes. Given that our initial parallel composition has name restrictions (νs) at the top level, $(\nu s)\, s\lightning \equiv \mathbf{0}$.

3.2 Affine multiparty session typing system

Global and local types

The advantage of affine session frameworks is that no change of the syntax of types from the original system is required. We follow [46] which is the most widely used syntax in the literature. A *global type*, written $G, G', \ldots$, describes the whole conversation scenario of a multiparty session as a type signature, and a *local type*, written by $S, S', \ldots$, represents a local protocol for each participant. The syntax of types is given as:

▶ **Definition 3.7** (Global types). The syntax of a **global type** G is:

$$G ::= \mathtt{p}\to\mathtt{q}\colon \{\mathtt{m}_i(S_i).G_i\}_{i\in I} \ \big|\ \mu\mathbf{t}.G \ \big|\ \mathbf{t} \ \big|\ \mathbf{end}$$

with $\mathtt{p}\neq\mathtt{q}$, $I\neq\emptyset$, and $\forall i\in I : \mathrm{fv}(S_i) = \emptyset$. The syntax of **local types** is:

$$S, T ::= \mathtt{p}\&_{i\in I}\mathtt{m}_i(S_i).S'_i \ \big|\ \mathtt{p}\oplus_{i\in I}\mathtt{m}_i(S_i).S'_i \ \big|\ \mathbf{end} \ \big|\ \mu\mathbf{t}.S \ \big|\ \mathbf{t}$$

with $I\neq\emptyset$, and $\mathtt{m}_i$ pairwise distinct.
Types must be closed, and recursion variables to be guarded.

$\mathtt{m} \in \mathbb{A}$ corresponds to the usual message labels in the session type theory. Global branching type $\mathtt{p}\to\mathtt{q}\colon \{\mathtt{m}_i(S_i).G_i\}_{i\in I}$ states that participant $\mathtt{p}$ can send a message with one of the $\mathtt{m}_i$ labels and a *message payload type* S_i to the participant $\mathtt{q}$ and that interaction described in G_i follows. We require $\mathtt{p} \neq \mathtt{q}$ to prevent self-sent messages and $\mathtt{m}_i \neq \mathtt{m}_k$ for all $i \neq k \in J$. Recursive types $\mu\mathbf{t}.G$ are for recursive protocols, assuming those type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e., they only occur under branching. Type $\mathbf{end}$ represents session termination (often omitted). We write $\mathtt{p} \in \mathrm{roles}(G)$ (or simply $\mathtt{p}\in G$) iff, for some $\mathtt{q}$, either $\mathtt{p}\to\mathtt{q}$ or $\mathtt{q}\to\mathtt{p}$ occurs in G. The function $\mathsf{id}(G)$ gives the participants of G.

For local types, the *branching type* $\mathtt{p}\&_{i\in I}\mathtt{m}_i(S_i).S'_i$ specifies the reception of a message from $\mathtt{p}$ with a label among the $\mathtt{m}_i$ and a payload S_i. The *selection type* $\mathtt{p}\oplus_{i\in I}\mathtt{m}_i(S_i).S'_i$ is its *dual* – its opposite operation. The remaining type constructors are as for global types. We say a type is *guarded* if it is neither a recursive type nor a type variable.

The relation between global and local types is formalised by projection [10, 21]. The *projection of* G *onto* $\mathtt{p}$ is written $G\upharpoonright\mathtt{p}$ and the standard subtyping relation, $\leqslant$. See [33].

We define typing contexts which are used to define properties of type-level behaviours.

▶ **Definition 3.8** (Typing contexts). *Θ denotes a partial mapping from process variables to n-tuples of types, and Γ denotes a partial mapping from channels to types, defined as:*

$$\Theta \ ::= \ \emptyset \ \mid \ \Theta,\, X{:}S_1,\dots,S_n \qquad \Gamma \ ::= \ \emptyset \ \mid \ \Gamma, c{:}S$$

The composition Γ_1, Γ_2 *is defined iff* $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. *We write* $s \notin \Gamma$ *iff* $\forall\mathtt{p} : s[\mathtt{p}] \notin \mathrm{dom}(\Gamma)$ *(i.e., session s does not occur in Γ). We write* $\mathrm{dom}(\Gamma) = \{s\}$ *iff* $\forall c \in \mathrm{dom}(\Gamma)$ *there is* $\mathtt{p}$ *such that* $c = s[\mathtt{p}]$ *(i.e., Γ only contains session s); and* $\Gamma \leqslant \Gamma'$ *iff* $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ *and* $\forall c \in \mathrm{dom}(\Gamma){:}\,\Gamma(c) \leqslant \Gamma'(c)$. *We write* $\Gamma \to \Gamma'$ *with* $\Gamma = \Gamma_0,\ s[\mathtt{p}]{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i(S_i).S'_i,\ s[\mathtt{q}]{:}\mathtt{p}\&_{j\in J}\mathtt{m}_j(T_j).T'_j$ *and* $\Gamma' = \Gamma_0,\ s[\mathtt{p}]{:}S'_i,\ s[\mathtt{q}]{:}T'_j$ *where types are defined modulo unfolding recursive types. We write* $\Gamma \to^* \Gamma'$ *for a transitive and reflexive closure of* $\to$*; and* $\Gamma \to$ *if there exists* Γ' *such that* $\Gamma \to \Gamma'$.

Next, we define typing context properties defined by its reduction.

We say Γ *is safe*, written $\mathrm{safe}(\Gamma)$, if $\varphi(\Gamma)$ for some safety property φ. Similarly, for *deadlock-freedom* ($\mathrm{df}(\Gamma)$) and *liveness plus* ($\mathrm{live}^+(\Gamma)$). See [33] for the definitions. The reader can refer to [46] for more explanations of the typing context properties.

▶ **Definition 3.9** (Typing judgement). The typing judgement for processes has the form:

$$\Theta \cdot \Gamma \vdash P \text{ (with } \Theta/\Gamma \text{ omitted when empty)} \tag{8}$$

and are defined by the typing rules in Figure 4 with the judgements for process variables and channels. For convenience, we type-annotate channels bound by process definitions and restrictions. Note that $\mathrm{end}(\Gamma)$ denotes that Γ only contains type **end**.

We explain each rule highlighting the new rules from [46].

(Affine) Branching/Selection. [T-&] and [T-⊕] are the standard rules for branching and selection, which can also type affine branching and selection. Note that the premise Γ in $\Theta \cdot \Gamma, y_i{:}S_i, c{:}S'_i \vdash P_i$ in [T-&] ensures that selecting one branch in the reduction rule defined by [C-Br] is sufficient for ensuring type soundness.

Try-Catch and Cancellation. [T-try] is typing a try process: we ensure P has a unique subject and catch block process Q has the same session typing (similar with branching). [T-cancel] generates a kill process at its declared session.

Kill process. [T-kill] types a kill process that appears during reductions: the cancellation of $s[\mathtt{p}]$ is broadcasting the cancellation to all processes which belong to session s.

Recursions. [T-def] and [T-call] are identical to those of [46].

Restriction. Processes are initially typed projecting a global type by [T-init], while running processes are typed by [T-ν] (see the proof of Theorem 3.12).

$$\frac{\Theta(X) = S_1,\dots,S_n}{\Theta \vdash X{:}S_1,\dots,S_n}\,[\text{T-}X] \quad \frac{S \leqslant S'}{c{:}S \vdash c{:}S'}\,[\text{T-sub}] \quad \frac{\forall i \in 1..n \quad c_i{:}S_i \vdash c_i{:}\mathbf{end}}{\text{end}(c_1{:}S_1,\dots,c_n{:}S_n)}\,[\text{T-end}] \quad \frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}}\,[\text{T-}\mathbf{0}]$$

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\&_{i\in I}\mathtt{m}_i(S_i).S_i' \quad \forall i\in I \quad \Theta\cdot\Gamma, y_i{:}S_i, c{:}S_i' \vdash P_i}{\Theta\cdot\Gamma,\Gamma_1 \vdash \dagger\, c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(y_i).P_i}\,[\text{T-}\&] \quad \frac{\Theta\cdot\Gamma_1 \vdash P_1 \quad \Theta\cdot\Gamma_2 \vdash P_2}{\Theta\cdot\Gamma_1,\Gamma_2 \vdash P_1 \mid P_2}\,[\text{T-}|]$$

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\oplus\mathtt{m}(S).S' \quad \Gamma_2 \vdash c'{:}S \quad \Theta\cdot\Gamma, c{:}S' \vdash P}{\Theta\cdot\Gamma,\Gamma_1,\Gamma_2 \vdash \dagger\, c[\mathtt{q}]\oplus\mathtt{m}\langle c'\rangle.P}\,[\text{T-}\oplus] \quad \frac{\Theta\cdot\Gamma\vdash P \quad \text{sbj}(P)=\{c\} \quad \Theta\cdot\Gamma\vdash Q}{\Theta\cdot\Gamma\vdash \mathbf{try}\ P\ \mathbf{catch}\ Q}\,[\text{ T-try }]$$

$$\frac{\text{end}(\Gamma) \quad 0\le n}{\Theta\cdot\Gamma, s[\mathtt{p}_1]{:}S_1,\dots,s[\mathtt{p}_n]{:}S_n \vdash s\lightning}\,[\text{ T-kill }] \quad \frac{\Theta\cdot\Gamma\vdash Q}{\Theta\cdot\Gamma, c{:}S \vdash \mathbf{cancel}(c).Q}\,[\text{ T-cancel }]$$

$$\frac{\Theta, X{:}S_1,\dots,S_n \cdot x_1{:}S_1,\dots,x_n{:}S_n \vdash P \quad \Theta, X{:}S_1,\dots,S_n\cdot\Gamma\vdash Q}{\Theta\cdot\Gamma\vdash \mathbf{def}\ X(x_1{:}S_1,\dots,x_n{:}S_n)=P\ \mathbf{in}\ Q}\,[\text{T-def}]$$

$$\frac{\Theta\vdash X{:}S_1,\dots,S_n \quad \text{end}(\Gamma_0) \quad \forall i\in 1..n \quad \Gamma_i\vdash c_i{:}S_i}{\Theta\cdot\Gamma_0,\Gamma_1,\dots,\Gamma_n \vdash X\langle c_1,\dots,c_n\rangle}\,[\text{T-call}]$$

$$\frac{\Gamma' = \{s[\mathtt{p}]{:}S_{\mathtt{p}}\}_{\mathtt{p}\in I} \quad s\notin\Gamma \quad \text{safe}(\Gamma') \quad \Theta\cdot\Gamma,\Gamma'\vdash P}{\Theta\cdot\Gamma\vdash(\nu s{:}\Gamma')\,P}\,[\text{T-}\nu]$$

$$\frac{\Gamma' = \{s[\mathtt{p}]{:}G\restriction\mathtt{p}\}_{\mathtt{p}\in\text{roles}(G)} \text{ or } \text{end}(\Gamma') \quad s\notin\Gamma \quad \Theta\cdot\Gamma,\Gamma'\vdash P}{\Theta\cdot\Gamma\vdash(\nu s{:}\Gamma')\,P}\,[\text{ T-init }]$$

Figure 4 Multiparty session typing rules. We highlight the new rules from [46].

▶ **Example 3.10** (Typing AMPST processes). To demonstrate the typing rules we type the inner *try* process from the reduction example. Let $Q = \mathbf{try}\ R\ \mathbf{catch}\ \mathbf{cancel}(s[\mathtt{p}]).\mathbf{0}$ where $R = {?}\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}$ and $\mathtt{d}$ is of type $S_1 = \mathbf{end}$. We show that $\Gamma\vdash Q$ where $\Gamma = \mathtt{d}{:}S_1, s[\mathtt{p}]{:}S_2$ with $S_2 = \mathtt{r}\oplus\mathtt{res}(S_1).\mathbf{end}$.

$$\dfrac{\dfrac{s[\mathtt{p}]{:}S_2\vdash s[\mathtt{p}]{:}S_2 \quad \mathtt{d}{:}S_1\vdash\mathtt{d}{:}S_1 \quad \dfrac{\cdots}{s[\mathtt{p}]{:}\mathbf{end}\vdash\mathbf{0}}[\text{T-}\mathbf{0}]}{\Gamma\vdash {?}\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}}[\text{T-}\oplus] \quad \text{sbj}(R)=\{s[\mathtt{p}]\} \quad \dfrac{\dfrac{\cdots}{\mathtt{d}{:}S_1\vdash\mathbf{0}}[\text{T-}\mathbf{0}]}{\Gamma\vdash\mathbf{cancel}(s[\mathtt{p}]).\mathbf{0}}[\text{T-cancel}]}{\Gamma\vdash Q}[\text{T-try}]$$

3.3 Properties of affine multiparty session types

This subsection proves the main properties of AMPST processes. We first prove basic properties such as Subject Congruence and Reduction Theorems, then prove important properties, session fidelity, deadlock-freedom and liveness. The highlight is cancellation termination, which guarantees that once an exceptional behaviour is triggered, all parties in a single session can terminate as nil processes.

Unlike linear-logic based typing systems [39], we do *not* assume that the typing system is closed modulo $\equiv$. Instead, we prove closedness of $\equiv$ for tricky cases, e.g., kill and try-catches.

▶ **Theorem 3.11** (Subject Congruence). If $\Theta\cdot\Gamma\vdash Q$ and $Q\equiv P$, then we have $\Theta\cdot\Gamma\vdash P$.

By Theorem 3.11, AMPST processes satisfy *type soundness*.

▶ **Theorem 3.12** (Subject Reduction). Suppose $\Theta\cdot\Gamma\vdash P$ and Γ safe. Then, $P\to P'$ implies there exists Γ' such that Γ' is safe and $\Gamma\to^*\Gamma'$ and $\Theta\cdot\Gamma'\vdash P'$.

A single agent in a multiparty session s is a participant playing a single role $\mathtt{p}$ in s. We use the definition from [46] except the highlighted part, which now includes affine processes.

▶ **Definition 3.13** (A unique role process). *Assume $\emptyset \cdot \Gamma \vdash P$. We say that P:*

1. ***has guarded definitions*** *iff in each subterm of the form* $\textbf{def}\ X(x_1{:}S_1,\ldots,x_n{:}S_n) = Q\ \textbf{in}\ P'$, *for all* $i \in 1..n$, $S_i \not\leqslant \textbf{end}$ *implies that a call* $Y\langle\ldots,x_i,\ldots\rangle$ *can only occur in* Q *as subterm of* $\dagger x_i[\mathtt{q}]\sum_{j\in J}\mathtt{m}_j(y_j).P_j$ *or* $\dagger x_i[\mathtt{q}]\oplus\mathtt{m}\langle c\rangle.P''$ *(i.e., after using* x_i *for selection/branching);*
2. ***only plays role*** $\mathtt{p}$ ***in*** s***, by*** Γ***,*** *iff:*
   - **i)** P *has guarded definitions;*
   - **ii)** $\mathrm{fv}(P) = \emptyset$*;*
   - **iii)** $\Gamma = \Gamma_0, s[\mathtt{p}]{:}S$ *with* $S \not\leqslant \textbf{end}$ *and* $\mathrm{end}(\Gamma_0)$*;*
   - **iv)** *in all subterms* $(\nu s'{:}\Gamma')\,P'$ *of* P*, we have* $\Gamma' = s'[\mathtt{p}']{:}\textbf{end}$ *(for some* $\mathtt{p}'$*).*

We say "P ***only plays role*** $\mathtt{p}$ ***in*** s*" iff* $\exists\Gamma : \emptyset\cdot\Gamma\vdash P$*, and item 2 holds.*

Note that by definition, a unique role process in s includes $s\lightning$.

Session fidelity is an important property to ensure liveness and deadlock-freedom, as well as termination. We extend that in [46] by taking a kill process into account. A set of unique role processes of a single multiparty session, together with kill processes always make progress if a typing context has progress, satisfying a protocol compliance.

Below we write $Q\lightning$ if Q contains only a parallel composition of kill processes.

▶ **Theorem 3.14** (Session Fidelity). *Assume* $\emptyset\cdot\Gamma\vdash P$*, where* Γ *is safe,* $P \equiv \big|_{\mathtt{p}\in I} P_{\mathtt{p}} \mid Q\lightning$*, and* $\Gamma = \bigcup_{\mathtt{p}\in I}\Gamma_{\mathtt{p}} \cup \Gamma_0$ *such that, for each* $P_{\mathtt{p}}$*, we have* $\emptyset\cdot\Gamma_{\mathtt{p}}\vdash P_{\mathtt{p}}$*; and* $\emptyset\cdot\Gamma_0\vdash Q\lightning$*. Assume that each* $P_{\mathtt{p}}$ *is either* $P_{\mathtt{p}} \equiv \mathbf{0}$*, or only plays* $\mathtt{p}$ *in* s*, by* $\Gamma_{\mathtt{p}}$*. Then,* $\Gamma\!\rightarrow$ *implies* $\exists\Gamma', P'$ *such that* $\Gamma\rightarrow\Gamma'$*,* $P\rightarrow^* P'$ *and* $\emptyset\cdot\Gamma'\vdash P'$*, with* Γ' *safe,* $P' \equiv \big|_{\mathtt{p}\in I} P'_{\mathtt{p}} \mid Q'\lightning$*, and* $\Gamma' = \bigcup_{\mathtt{p}\in I}\Gamma'_{\mathtt{p}} \cup \Gamma'_0$ *such that, for each* $P'_{\mathtt{p}}$*, we have* $\emptyset\cdot\Gamma'_{\mathtt{p}}\vdash P'_{\mathtt{p}}$*, and each* $P'_{\mathtt{p}}$ *is either* $\mathbf{0}$*, or only plays* $\mathtt{p}$ *in* s*, by* $\Gamma'_{\mathtt{p}}$*; and* $\emptyset\cdot\Gamma'_0\vdash Q'\lightning$*.*

By the above theorem, we can prove deadlock-freedom and liveness for a single session multiparty session in the presence of affine processes.

▶ **Definition 3.15** (Deadlock-freedom and liveness).

1. P is **deadlock-free** iff $P\rightarrow^* P' \not\rightarrow$ implies $P' \equiv \mathbf{0}$.
2. P is **live** iff $P\rightarrow^* P' \equiv \mathbb{C}[Q]$ implies:
   - **i)** if $Q = c[\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.Q'$ (for some $\mathtt{m}, s', \mathtt{r}, Q'$), then $\exists\mathbb{C}'$: $P'\rightarrow^*\mathbb{C}'[Q']$; and
   - **ii)** if $Q = c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).Q'_i$ (for some $\mathtt{m}_i, x_i, Q'_i$), then $\exists\mathbb{C}', k\in I, s', \mathtt{r}$: $P'\rightarrow^* \mathbb{C}'[Q'_k\{^{s'[\mathtt{r}]}/_{x_k}\}]$.

Note that liveness is defined for *linear* selection or *linear* branching processes which appear at the top level, i.e., under the reduction context $\mathbb{C}$, not under `try-catch` construct, cancel nor affine branching and selection processes.

▶ **Theorem 3.16** (Deadlock-freedom). *Assume* $\emptyset\cdot\Gamma\vdash P$*, with* Γ *safe,* $P \equiv \big|_{\mathtt{p}\in I} P_{\mathtt{p}}$*, each* $P_{\mathtt{p}}$ *either* $P_{\mathtt{p}} \equiv \mathbf{0}$*, or only playing role* $\mathtt{p}$ *in* s*. Then,* $\mathrm{df}(\Gamma)$ *implies that* $(\nu\tilde{s}{:}\Gamma)\,P$ *with* $\{\tilde{s}\} = \mathrm{dom}(\Gamma)$ *is deadlock-free.*

As discussed in [46, Definition 5.11], we require $\mathrm{live}^+(\Gamma)$ for proving liveness.

▶ **Theorem 3.17** (Liveness). *Assume* $\emptyset\cdot\Gamma\vdash P$*, with* Γ *safe,* $P \equiv \big|_{\mathtt{p}\in I} P_{\mathtt{p}}$*, each* $P_{\mathtt{p}}$ *either* $P_{\mathtt{p}} \equiv \mathbf{0}$*, or only playing role* $\mathtt{p}$ *in* s*. Then,* $\mathrm{live}^+(\Gamma)$ *implies that* P *is live.*

Now we consider a user-written Rust program with one session as an *initial program.*

▶ **Definition 3.18** (Initial program). We say $\vdash Q$ is an *initial program* if

1. $Q \equiv (\nu \tilde{s}{:}\Gamma) \big|_{\mathtt{p}\in G} P_{\mathtt{p}}$ with $\{\tilde{s}\} = \mathrm{dom}(\Gamma)$;
2. $P_{\mathtt{p}}$ only plays $\mathtt{p}$ in s;
3. in each subterm of the form, **def** $X(\widetilde{x}) = Q$ **in** P', (1) Q is of the form **try** Q' **catch** P''; and (2) P'' does not contain any (free or bound) process call.
4. $\Gamma = \{s[\mathtt{p}]{:}G{\upharpoonright}\mathtt{p}\}_{\mathtt{p}\in G}, \Gamma'$ for some G and $\mathrm{end}(\Gamma')$;
5. $\vdash Q$ is derived using [T-init] instead of [T-ν]; and without [T-kill].

Condition (3) ensures that once a process moves to the *catch-block*, then it ensures finite computation; (4,5) state that the initial program starts conforming to a global protocol.

▶ Remark 3.19 (Initial processes). Condition (3) does not limit the expressiveness since the *try-block* can include infinite computations; and conditions (4,5) imply that an initial program typed by condition (1) has started. Notice that running (runtime) processes generated from the initial program are typed using [T-ν] and [T-kill]; hence the proof of the subject reduction holds with Lemma 3.20 below.

Before proving the main theorems, we state that a set of local types projected from a well-formed global type satisfy the safety property.

▶ **Lemma 3.20** ([46, Lemma 5.9]). *Let* $\Gamma = \{s[\mathtt{p}]{:}G{\upharpoonright}\mathtt{p}\}_{\mathtt{p}\in\mathrm{roles}(G)}$. *Then* $\mathrm{safe}(\Gamma)$, $\mathrm{df}(\Gamma)$ *and* $\mathrm{live}^{+}(\Gamma)$.

Now we state the two main theorems of this paper: deadlock-freedom, liveness and cancellation termination. The cancellation termination theorem states that once a kill signal is produced by cancellation or affine processes (due to a timeout or an error), then all processes are enabled to terminate. We start from deadlock-freedom.

▶ **Corollary 3.21** (Deadlock-freedom and liveness for an initial program). Suppose $\vdash Q$ is an initial program. Then for all P such that $Q \to^* P$, P is deadlock-free and live.

▶ **Theorem 3.22** (Cancellation Termination). *Suppose* $\vdash Q$ *is an initial program. If* $Q \to^* \mathbb{C}[s\lightning] = P'$, *then we have* $P' \to^* \mathbf{0}$.

▶ **Corollary 3.23** (Cancellation Termination of Affine and Cancel Processes). *Suppose* $\vdash Q$ *is an initial program.*

1. *If* $Q \to^* \mathbb{C}[\textbf{\textit{cancel}}(s[\mathtt{p}]).Q'] = P'$, *then we have* $P' \to^* \mathbf{0}$.
2. *If* $Q \to^* \mathbb{C}[\mathbb{E}[\,?\,s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i]] = P'$ *or* $Q \to^* \mathbb{C}[\mathbb{E}[\,?\,s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P]] = P'$, *then we have* $P' \to^* \mathbf{0}$.

▶ Remark 3.24 (Termination theorem). The cancellation termination theorem means that *there always exists a path* which leads to $\mathbf{0}$; and an initial program might not terminate even if it contains a process with $s\lightning$. This differs from the total termination, i.e., all paths are finite – a program will definitely stop as $\mathbf{0}$. However, if we apply *fair traversal sets*, i.e., fair scheduling, from [46, Definition 5.5], applying to processes in $\mathbb{C}[s\lightning]$, we can prove the total termination. Since these extensions require an introduction of labelled transition systems for processes, we leave it as future work.

4 Design and implementation of `MultiCrusty`

4.1 Challenges for the implementation of `MultiCrusty`

The three main challenges underpinning the implementation of AMPST in Rust are related to multiparty communications and ensuring correctness for affine channels.

```
1 pub struct MeshedChannels< S1: Session, S2: Session, R: Role, N: Role> {
2 session1: S1, session2: S, stack: R, name: N }
```

Figure 5 Generated MeshedChannles structure.

(Challenge 1) Realising a multiparty channel by binary channels. AMPST relies on a *multiparty channel* – a channel that can communicate with several roles. In Rust, communication channels are peer-to-peer, e.g., they are *binary* [30]. To overcome this limitation, we extend an encoding of MPST into binary channels [44]. In this encoding, a multiparty channel can be represented as an *indexed tuple* of *one-shot binary channels* used in a sequence depending on the ordering specified by the type. This design ensures reception error safety by construction. Since each pair of binary channels is dual, then no communication mismatch can occur. We piggyback on this result by introducing meshed channels, which reuse an existing library of binary session types in Rust [30] with built-in duality guarantees. We explain the implementation of meshed channels in § 4.2. See [33] for usecases that demonstrate how to use `MultiCrusty` for programming distributed protocols.

(Challenge 2) Deadlock-freedom, liveness and termination. Duality is unfortunately insufficient to guarantee deadlock-freedom. The naive decomposition of binary channels leads to *hard to* detect deadlock errors [44]. To ensure liveness properties and correct termination of cancellation behaviour, we integrate `MultiCrusty` with two state-of-the-art verification toolchains – Scribble [27] and k-MC [35], that ensures meshed channel types are *correct*. The former generates correct meshed channel types in Rust, while the latter verifies a set of existing meshed channel types. In both cases, well-typed processes implemented using well-typed meshed channels are free from deadlocks, orphan messages and reception errors. We display the Rust types for our running example in § 4.3.

(Challenge 3) Affinity with `try-catch` and optional types. Rust does not have a native `try-catch` construct, but macros and optional types. We use them to design and implement a `try-catch` block and affine selection and branching. Channels can be implicitly or explicitly cancelled, and all processes are guaranteed to terminate gracefully in the event of a cancellation, avoiding endless cascading errors. We discuss our design choices in § 4.4.

4.2 Meshed Channels in `MultiCrusty`

A multiparty channel in `MultiCrusty` is realised as an *affine meshed channel* (hereafter meshed channel), which has three ingredients: (1) a list of separate binary channels (one binary channel for each pair of participants); (2) a stack that imposes the order between the binary channels; and (3) the name of the role, whose behaviour is implemented by the meshed channel. Figure 5 shows a generated meshed channel when using the macro `gen_mpst!(MeshedChannels, A, C, S)` for a 3-party protocol.

The generated structure, `MeshedChannels`, holds four fields. The first two fields, `session1` and `session2`, are of type `Session` which is a binary session type. Therefore, these fields store binary channels. `Session` in Rust is a `trait` and a `trait` is similar to an interface. The `Session` trait can be instantiated to three generic (binary session) types: an `End` type; a `Recv<T, S>` or a `Send<T, S>` type, with their respective payload of type `T` and their continuation of a binary session type `S`. This has important implications for the design and safety of our system. Since all pairs of binary channels are created and distributed across meshed channels at the

```
1   // Declare the name of the role
2   type NameA = RoleA<RoleEnd>;
3   // Binary session types for A and C
4   type AtoCVideo<N> = Recv<N, Send<N, Recv<ChoiceA<N>, End>>
5   // Binary session types for A and S
6   type AtoSVideo<N> = Send<N, Recv<N, End>>;
7   // Declare usage order of binary channels inside a meshed channel
8   type StackAInit = RoleC<RoleEnd>; // for the initial meshed channel
9   type StackAVideo = RoleC<RoleS<RoleS<RoleC<RoleEnd>>>>; // for branch Video
10  // Declare the type of the meshed channel
11  type RecA<N> = MeshedChannels<Recv<ChoiceA<N>, End>, End, StackAInit, NameA>;
12  // Declare an enum with variants corresponding to the different branches, \ie Video and End
13  enum ChoiceA<N> {
14   Video(MeshedChannels<AtoCVideo<N>, AtoSVideo<N>, StackAVideo, NameA>),
15   Close(MeshedChannels<End, End, RoleEnd, NameA>)
16  }
```

Figure 6 Local Rust types for role A (Authenticator) from Figure 2b.

start of the protocol, the binary type `Session` enforces that each pair of binary channels are dual. For example, the binary channel for role S inside the meshed channels for role A; and the binary channel for role A inside the meshed channels for role S are dual. This design ensures that, without using any external tools, our system is communication safe, no reception error can occur. This is insufficient to guarantee deadlock-freedom, which is why we utilise Scribble or bounded model checking, i.e., k-MC, as an additional verification step.

The rest of the fields of the `struct MeshedChannels` are stack-like structures, `stack` and `name`, which represent respectively the order of the interactions (in what order the binary channels should be used) and the associated role. For instance, the behaviour where role A has to communicate first with role S, then with role C and then the session ends, can be specified using a stack of type `RoleS<RoleC<RoleEnd>>`. Note that all stack types such as `RoleS` and `RoleC` are generated singleton types. Role names are codified as `RoleX<RoleEnd>` where `X` is the actual name of the participant. For instance, role A is realised as the singleton type `RoleA<RoleEnd>`. We chose this design for its readability and its ease of implementation: one can guess at a glance the current state of a participant.

The code generation macro `gen_mpst!` produces *meshed channels for any finite number of communicating processes*. For example, in the case of a protocol with four roles, the macro `gen_mpst!` will generate a meshed channel with five fields – one field for the binary session between each pair of participants (which is 3 fields in total), one field for the stack and one field for the name of the role that is being implemented.

4.3 Types for affine meshed channels

Meshed channel types – `MeshedChannels` – correspond to local session types. They describe the behaviour of each meshed channel and specify which communication primitives are permitted on a meshed channel. To better illustrate meshed channel types, we explain the type `RecA<N>` for role A (Authenticator) from Figure 2b. The types are displayed in Figure 6. The types of the meshed channels for the other roles, i.e., C and S are available in [33].

Following the protocol, the first action on A is an external choice. Role A should receive a choice from role C of either `Video` or `Close`. External choice is realised in `MultiCrusty` as an `enum` with a variant for each branch, where each variant is parameterised on the meshed channel that will be used for that branch. The enum type `ChoiceA<N>` in line 13 precisely specifies this behaviour – two variants with their respective meshed channels. The branch `Close` is trivial since no communication apart from closing all channels is expected in this branch. Hence, the binary channels for S and A, and C and A are all `End`. The type of the

meshed channel for the branch `Video` in line 14 is more elaborate. `MeshedChannels<AtoCVideo<N>, AtoSVideo<N>, StackAVideo, NameA>` specifies that the type of the binary channel for C and A is `AtoCVideo<N>`, the type of the binary channel for role S and role A is `AtoSVideo<N>`, the stack of the meshed channel is `StackAVideo`. The declaration `RoleC<RoleS<RoleS<RoleC<RoleEnd>>>>` specifies the order in which binary channels must the used – first the binary channel with C, then with role S, then with S again, and finally with C. The last argument specifies that this is a meshed channel for role A.

The meshed channel types can be written either by the developers and verified using an external tool, k-MC, or generated from a global protocol written in Scribble.

4.4 Exception and cancellation

Exception handling

Rust does not have exceptions. Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error. `Result<T, E>` is a variant type with two constructors: `Ok(T)` and `Err(E)` where `T` and `E` are generic type parameters.

We leverage two mechanisms to implement the semantics presented in § 3, both of which rely on the `Result` variant type: (1) the `?` operator and (2) the `attempt!-catch` macro. The `?` is syntactic sugar for error message propagation. More specifically, each communication primitive is wrapped inside a `Result` type. For example, the return type of `recv()` is `Result<(T, S), Box<dyn Error>>`. The call `recv()` on the multiparty channel `s` triggers the attempt of the reception of a tuple containing a payload of type `T` and a continuation of type `S`.

If a peer tries to read a cancelled endpoint then an error message is returned. Therefore, if an error occurs during receive due to, for example, the cancellation of the other end of the channel, the `?` operator stops the `recv()` function and returns an `Err` value to the calling code. Then, the user can decide to handle the error or `panic!` and terminate the program.

Similarly, the `attempt!-catch` block is syntactic sugar that allows exception handling over multiple communication actions. For instance, the `attempt! M catch N` reduces to its failing clause `N` if an error occurs in any of the statements in `M`. The interested users can try the online Rust playground that demonstrates the implementation of `attempt!-catch` using the `and_then` combinator [13]. The `attempt! M catch N` corresponds to the `try-catch` in § 3.

The implementation follows the behaviour formalised by the reduction rules in § 3. In particular, it ensures that whenever an error happens, a session is cancelled ($s⚡$). We utilise Rust drop mechanism. When a value in Rust goes out of scope, Rust automatically drops it by calling its destructor: the `Drop` method. A variable that cannot be cloned, such as a session `s`, is out scope when used in a function and not returned, such as when used in the `close()` and `cancel()` functions. We have customised this method by implementing the `Drop` trait, which explicitly calls `cancel()`. If an error occurs, and the meshed channel is not explicitly cancelled, the meshed channel is *implicitly cancelled* from its destructor. In the case of a `panic!`, the session `s` will be dropped, alongside all variables within the same function, when `panic!` is called. Similarly to the theory, `cancel(s)` is not mandatory and can be placed arbitrarily within the process. Calling `cancel(s)` is mostly used for expressiveness and mock tests purposes, when a failure, without `panic!`, needs to be simulated.

Session Cancellation

We discuss all cases involving session cancellation below:

1. **Implicit vs explicit cancellation.** Receiving on or closing disconnected sessions returns an error. As a result of the error, the multiparty channel `s` is cancelled by our underlying library, and all binary channels associated with `s` are disconnected. We call this an implicit cancellation. This behaviour implements rules [C-?Sel] and [C-?Br]. Alternatively, the user can also cancel the session explicitly.
2. **Raising an exception.** An error occurs (1) as a result of a communication over a closed/cancelled channel, (2) as a result of a timeout on a channel, or (3) in case of an error in the user code. For example the function `get_video()` can return an error. Then the user can decide to (1) `cancel(s)` the session, (2) silently drop the session, or (3) proceed with the protocol. Even if the user does not explicitly call the `cancel(s)` primitive, Rust runtime ensures that the meshed channel is always cancelled in the end.
3. **Double cancellation.** If a peer tries to cancel a session `s` that is already cancelled from another endpoint, then the cancellation is ignored. Note that in our semantics this behaviour is modelled using the structural congruence rules, namely $s\lightning \mid s\lightning \;\equiv\; s\lightning$.
4. **Cancel propagation.** When a session is cancelled, no communication action can be used subsequently on that channel. The action `cancel(s)` cancels all binary channels that are a part of the meshed channel, which precisely simulates the kill process $s\lightning$. When a peer attempts to receive on a channel, if either side of the channel is cancelled, the operation returns an error, and the session in scope is dropped. This is exactly the behaviour for the channels from the `crossbeam-channel` library, and we inherit and extend this behaviour to our library. Since our *receive* happens on a binary channel, our extension ensures that all other binary channels that are in scope, and the ones that are in the stack, are also closed. Since these channels are closed, when other peers try to read from them, they will also encounter an error, and will subsequently close their channels.

5 Evaluations: benchmarks, expressiveness and case studies

We evaluate `MultiCrusty` in terms of run-time performance (**§ 5.1**), compilation time (**§ 5.1**) and applications (**§ 5.2**, see [33]). Through this section, we demonstrate the applicability of `MultiCrusty` and compare its performance with programs written in binary sessions and untyped implementations (Bare) using `crossbeam-channel`. The purpose of the microbenchmarks is to demonstrate the best and worst-case scenarios for the implementation: we have not considered performance as a primary consideration in the current implementation. The results show that rewriting multirole protocols from binary channels to affine meshed channels can have a performance gain in addition to the safety guarantees provided by MPST.

In summary, `MultiCrusty` has only a negligible overhead when compared to the built-in unsafe Rust channels, provided by `crossbeam-channel`, and up to two-fold runtime improvement to binary sessions in protocols with high-degree of synchronisation. The source files of the benchmarks and a script to reproduce the results are included in the artifact.

5.1 Performance

The goal of the microbenchmarks is two-fold. On one hand, it provides assurance that `MultiCrusty` does not incur significant overhead when compared to alternative libraries. The source of the runtime overhead of `MultiCrusty` can be attributed to: (1) the additional data structures that are generated (see § 4.2); and (2) checks for cancellation (as outlined in § 4.4). We also evaluate the efficiency of `MultiCrusty` when implementing multiparty (as opposed to binary) protocols. Multiparty protocols specify interaction dependencies between multiple threads. It is well-understood that a naive decomposition of multiparty protocol to

name	ping-pong	ring	full-mesh
diagram	ping A1 A2 pong	ping ping ping A1 A2 A3 An pong pong pong	A pong ping ping pong B pong ping C

Figure 7 Protocols for Microbenchmarks.

a binary one (without preserving interaction dependencies) not only causes race conditions and wrong results but also deadlocks [44]. One may mitigate this problem by utilising a synchronisation mechanism, which is an off-the-shelf alternative to meshed channels. We compare the performance of `MultiCrusty` and meshed channels to a binary-channels-only implementation that uses thread-synchronisation.

We compare implementations, written using (1) `MultiCrusty` API (MPST) without cancellation; (2) `MultiCrusty` API with cancellation (AMPST); (3) binary channels, following [30] (BC); and (4) a Bare-Rust implementation (Bare) using untyped channels as provided by the corresponding transport library `crossbeam-channel`. As a reminder, `MultiCrusty` uses [30]'s channels (which are binary only and technically non-meshed), and [30]'s channels use `crossbeam-channel` for actually sending and receiving payloads: the scaffolding of all programs differs only in the final communication primitives used. In addition, the BC implementations synchronise between threads when messages must be received in order.

Figure 7 shows simple visualisation, displayed for illustrative purpose, of the three examples that we benchmark. Figure 8 reports the results on runtime performance, i.e., the time to complete a protocol by the implemented endpoints in Rust, and compilation time, i.e., the time to compile the implementations for all roles. We stress tested the library up to 20 participants but only show the results up to 10 participants for readability.

Setup: Our machine configurations are AMD Opteron™ Processor 6282 SE @ 1.30 GHz with 32 cores/64 threads, 128 GB of RAM and 100 GB of HDD with Ubuntu 20.04, and with the latest version available for Rustup (1.24.3) and the Rust cargo compiler (1.56.0). We use *criterion* [29], a popular benchmark framework in Rust. We repeat each benchmark 10000 times and report the average execution time with a fairly narrow confidence interval of 95%.

Ping-pong

benchmark measures the execution time for completing a recursive protocol between two roles repeatedly increasing the number of executions for request-response unit messages. Figure 8a displays the running time w.r.t. the number of iterations. This protocol is binary, and this benchmark measures the pure overhead of MPST implementation. MPST directly reuses the BC library, adding the structure `MeshedChannels` on top of it. Since both implementations need the same number of threads, the benchmark compares only the overhead of `MeshedChannels`. Both MPST and AMPST have a linear performance increase compared to BC and Bare. MPST is about 2.5 times slower than BC and about 6.5 times slower than Bare for 500 iterations.

Ring

protocol, as seen in Figure 7, specifies `N` roles, connected in a ring, sending one message in a sequence. This example is sequential and stress tests the usage of numerous binary channels in an `MultiCrusty` implementation. Figure 8b displays the running time w.r.t.

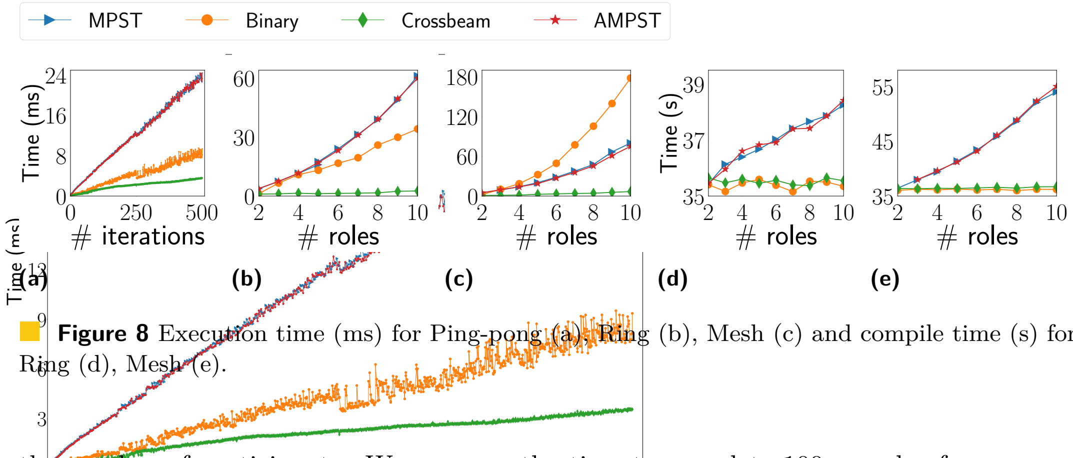


Figure 8 Execution time (ms) for Ping-pong (a), Ring (b), Mesh (c) and compile time (s) for Ring (d), Mesh (e).

the number of participants. We measure the time to complete 100 rounds of a message for an increasing number of roles. This benchmark demonstrates a worst-case scenario for `MultiCrusty` since the MPST implementation requires `N*N` binary channels, hence `N*N` interactions at most, meanwhile the other implementations only need `2*N` binary channels. `MultiCrusty` is increasingly slower than the other implementations following a quadratic curve. All the implementations are running at the same speed for 2 participants; MPST becomes almost 2 times slower than BC for 10 participants and almost 3.25 times slower for 20 participants. AMPST implementation has a negligible overhead compared to MPST.

Full-mesh

benchmark measures the execution time for completing a recursive protocol between `N` roles mutually exchanging the same message together: for every iteration, each participant sends and receives once with every other participant. For simplicity, we show the pattern in Figure 7 for three roles only. Figure 8c displays the running time w.r.t. the number of participants. This is a best-case scenario protocol for `MultiCrusty` since the protocol requires a lot of explicit synchronisation if implemented as a composition of binary protocols. The slowdown of BC is explained by the difference of implementation and the management of threads: the `MultiCrusty` needs only one thread for each participant, meanwhile for the binary case, two threads per pair of interactions are required to ensure that the message causalities are preserved. All implementations have similar running time for 2 participants but MPST is about 2.3 times faster than BC, and about 11 times slower than Bare for 10 participants. The figure only displays the results for up to 10 participants, since this is sufficient to show the overhead trend. In practice, we measured for up to 20 participants. For reference, at 20 participants, MPST is about 12 times slower than Bare and about 3.75 times faster than BC. As expected, AMPST has almost the same running time as MPST.

Results summary on execution time

Overall, `MultiCrusty` is faster than the BC implementation when there are numerous interactions and participants, thanks to the encapsulation of each participant as a thread; the worst-case scenario for `MultiCrusty` is for protocols with many participants but no causalities between them which results in a slowdown when compared with BC. AMPST adds a negligible running time due to the simple checking of the status of the binary channels.

Results summary on compilation time

We also compare the compilation time of the three protocols using `cargo build`. The results are presented in Figures 8d and 8e. As expected, the more participants there are, the higher the compilation time for MPST, with up to 40% increase for the full-mesh protocol and only 11% for the ring protocol. We omit the graph for the ping-pong protocol since the number of iterations does not affect compilation time and the number of generated types, hence the compilation stays constant at 36.4s (MPST), 36.6s (AMPST), 36.1s (BC) and 36.3s (Bare).

The compilation time of BC and Bare are very close thanks to Rust's features, a mechanism to express conditional compilation and optional dependencies. This allows compiling only specific parts of libraries, instead of the whole libraries, depending on the needs of each file. For BC and Bare, we only compile `MultiCrusty`'s *default* features, meanwhile for MPST and AMPST, we also compile the *macros* features, which include heavy blocks of code and new dependencies for the creation of the new roles, meshed channels and associated functions.

5.2 Expressiveness

We demonstrate the expressiveness and applicability of `MultiCrusty` by implementing protocols for a range of applications. We also draw the examples from the session types literature, well-established *application protocols* (OAuth, SMTP), and distributed protocols (logging, circuit breaker). Protocols with more than 5 participants are not considered since having one global protocol with more participants can quickly become intractable in terms of protocol logic and is considered bad practice. The global protocols and patterns in the literature that have many participants are parameterised [6], participants can be grouped in kinds having the same type. Thereby, this will avoid a combinatorial explosion.

Table 2 displays the examples and related metrics. In particular, we report compilation time (Check./Comp./Rel.), execution time (Exec. Time), the number of lines of code (LoC) for implementing all roles in `MultiCrusty`, the lines of code generated from Scribble (Gen Types) and the total lines of code (All); the two following columns indicate whether the protocol involves three participants or more (MP), and if the protocol is recursive (Rec).

We report three compilation times corresponding to the different compilation options in Rust – `cargo check` which only type checks the code without producing binaries, `cargo build` which compiles the code with binaries and `cargo build -release` which, in addition, optimises the compiled artifact. Each recursive protocol is built/checked 100 times, and we display the average in the table. All protocols are type-checked within 27 seconds, while the basic compilations range between 36s and 41s and the optimised compilations vary between 80s and 97s. Those results represent the longest time we can expect for the respective build/check: Rust compilation is iterative, therefore, the usual compilation time should be shorter. A 30 seconds pause is short enough to not break the *flow* [9] of the mental headspace focused on the current task. Building the binaries takes longer, because of two heavy libraries used by `MultiCrusty` (`tokio` [8] and `hyper` [48]). The execution time of the protocols is measured by implementing only the communication aspects of the protocol, and orthogonal computation-related aspects are omitted. The execution time is the time to complete all protocol interactions, and even for larger protocols, it is negligible.

Table 2 does not contain protocols with more than 5 distinct participants because, in our experience, whenever more participants are needed, the protocol is parameterised [6]. We leave such extension for future investigation.

Table 2 Selected examples from the literature.

Example (Endpoint)	Check./Comp./Rel./Exec. Time	LoC Impl.	Gen Types/All	MP	Rec
Three buyers [28]	26.7s / 37.1s / 81.3s / 568 μs	143	37 / 180	✓	✓
Calculator [22]	26.5s / 36.9s / 81.3s / 467 μs	136	32 / 168	✗	✗
Travel agency [24]	26.5s / 37.6s / 84.8s / 8 ms	200	47 / 247	✗	✓
Simple voting [22]	26.3s / 36.7s / 82.4s / 396 μs	207	61 / 268	✗	✗
Online wallet [42]	26.4s / 37.8s / 84.4s / 759 μs	231	76 / 307	✗	✓
Fibonacci [22]	26.6s / 36.7s / 80.9s / 9 ms	141	23 / 164	✗	✓
Video Streaming service (**§ 2**)	26.3s / 37.4s / 83.0s / 11 ms	104	39 / 143	✓	✓
oAuth2 [42]	26.4s / 37.5s / 83.2s / 12 ms	215	61 / 276	✓	✓
Distributed logging ([33])	26.5s / 36.8s / 82.6s / 5 ms	252	59 / 311	✗	✓
Circuit breaker ([33])	26.5s / 38.5s / 87.0s / 18 ms	375	142 / 517	✓	✓
SMTP [15]	26.4s / 41.1s / 97.3s / 5 ms	571	143 / 714	✗	✓

6 Related work and future work

A vast amount of session types implementations based on theories exist, as detailed in the recent surveys on language implementations [1] and tools [17]. We discuss closely related works, focusing on (1) session types implementations in Rust (§ 6.1); (2) MPST top-down implementations (including other programming languages) (§ 6.2). For related work about Affine types and exceptions/error handling in session types, see [33].

6.1 Session types implementations in Rust

Binary session types (BST) have been implemented in Rust by [27], [30] and [7], whereas, to our best knowledge, [11] is the only implementation of multiparty session types in Rust.

[27] implemented binary session types, following [20], while [30] based their library on the EGV calculus by [16] (See [33]). Both verify at compile-time that the behaviours of two endpoint processes are *dual*, i.e., the processes are compatible. The latter library allows to write and check session typed communications, and supports exception handling constructs. Rust originally did not support *recursive types* so [27] had to use *de Bruijn* indices to encode recursive session types, while [30] uses Rust's native recursive types but only handles failure for `recv()` actions: according to [30], this is generally the case with asynchronous implementations. This is because once an endpoint has received several messages, it makes sense to cancel them at the receiver rather than the sender. In fact, raising an exception on a send operation in an asynchronous calculus actually breaks confluence.

The library by [27] relies on an older version of Rust, hence we build `MultiCrusty` on top of [30]. Notice that we formalised AMPST guaranteeing the MPST properties of `MultiCrusty` (such as deadlock-freedom, liveness and cancellation termination), which are not present in [30]. In addition, our benchmarks confirmed that, in protocols where most of the participants mutually communicate, `MultiCrusty` is up to two times faster than [30].

[7] introduces their library, Ferrite, that implements BST in Rust, adopting *intuitionistic logic-based typing* [5]. The library ensures *linear* typing of channels, and includes a recently shared name extension by [2], but cannot statically handle prematurely dropped channel endpoints. Since Ferrite lacks an additional causal analysis for ensuring deadlock-freedom by [3], deadlock-freedom and liveness among more than two participants are not guaranteed, unlike `MultiCrusty`. Ferrite also lacks documentation and tests, making it hard to use.

[14] presents an implementation of a library for programming typestates in Rust. The library ensures that Rust programs follow a typestate specification. The tool, however, has several limitations. Differently than other works on typestates (e.g., typestates in Java [31]), [14] implements and verifies only binary non-recursive protocols, without a static guarantee that all branches are exhaustively implemented.

[11] implements MPST using `async` and `await` primitives. Their main focus is a performance analysis of asynchronous message reordering and comparisons of their asynchronous subtyping algorithm with existing tools, including the k-MC tool [35]. Their algorithm is a sound approximation of the (undecidable) asynchronous subtyping relation [18], by which their tool enables to check whether an unoptimised (projected from a global type) CFSM and its optimised CFSM are under the subtyping relation or not. The main disadvantage of [11] is that their library depends on external tools for checking not only deadlock-freedom, but also communication-safety. Differently, `MultiCrusty` can guarantee *dual compatibility* (inherited from [30]) in a multiparty protocol, based on our meshed channels implementation.

Note that all the above implementations, but [11], are limited to *binary* and no formalism is proposed in their papers (see Table 3). Unlike `MultiCrusty`, neither failure handling nor cancellation termination is implemented or formalised in any of the above-mentioned works.

6.2 Multiparty session types implementations in other languages

We compare implementations of (top-down) MPST, ordered by date of publication, in Table 3, focusing on statically typed languages: we exclude MPST implementations by runtime monitoring such as Erlang [41] and Python [12].

The table is composed as follows, row by row:

Languages lists the programming languages introduced or used.

Mainstream language states if the language is broadly used among developers or not.

MPST top-down characterises the framework: Multiparty session types (MPST) or binary session types (BST). If the implementation allows the user to write MPST global types, it is called a top-down approach.

Linearity checking describes whether the linear usage of channels is not checked, checked at compile-time (*static*) or checked at runtime (*dynamic*).

Exhaustive choices check indicates whether the implementation can *statically* enforce the correct handling of potential input types. ✗ denotes implementations that do not support pattern-matching to carry out choices (branching) using switch statements on `enum` types.

Formalism defines the theoretical foundations of the implementations, such as (1) the end point calculus (the π-calculus (noted as π-cal.) or FJ [25]); (2) the (global) types formalism without any endpoint calculi (no typing system is given, and no subject reduction theorem is proved); or (3) no formalism is given (no theory is developed).

Communication safety outlines the presence or the absence of session type-soundness demonstration. Four languages, marked as △, provide the type safety only at type level. ✗$^{\bullet}$ means that the theoretical formalism does not provide linear types, therefore only type safety of base values is proved.

Deadlock-freedom is a property guaranteeing that all components are progressing or ultimately terminate (which correspond to deadlock-freedom in MPST). Four languages marked by △ proved deadlock-freedom only at the type level. ✓$^{\bullet}$ implies the absence of a formal link with the local configurations reduced from the projection of a global type. [1]

Liveness is a property which ensure that all actions are eventually communicated with other parties (unless killed by an exception in the case of AMPST).

[1] [19] did not prove that any typing context reduced from a projection of a well-formed global type satisfies a safety property (a statement corresponding to Lemma 3.20). Hence, deadlock-freedom is not provided for processes initially typed by a given global type. Note that their typing contexts contain new elements not found in those defined in [23], which weakens the link with the top-down approach.

Table 3 MPST top-down implementations.

	[30, 27, 7]	[43]	[22, 23]	[32]	[44]	[40]	[6]	[26]	[38]	[54]	[19]	[52]	[11]	`MultiCrusty`
Language	Rust	MPI-C	Java	Java	Scala	F#	Go	OCaml	Typescript	F*	EnsembleS	Scala	Rust	Rust
Mainstream language	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
MPST Top-Down	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Linearity check	static	✗	dynamic	✗	dynamic	dynamic	dynamic	static	static	static	static	dynamic	static	static
Exhaustive choices check	✓	✗	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓
Formalism	✗	✗	types	FJ	π-cal.	✗	types	π-cal.	types	types	π-cal.	π-cal.	types	π-cal.
Communication safety	✗	✗	△	✓	✓	✗	△	✗•	△	△	✓	✓	△	✓
Deadlock freedom	✗	✗	△	✗	✓	✗	△	✗	△	△	✓•	✓	△	✓
Liveness	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Cancellation termination	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Cancellation termination: once a cancellation happens at one of the participants in a multiparty protocol, the cancellation is propagated correctly, and all processes can terminate.

The Rust implementations in the first column of Table 3 are included for reference.

Most of the MPST implementations [23, 44, 40, 6, 38, 54] follow the methodology given by [22], which generates Java communicating APIs from Scribble [53, 47]. They exploit the equivalence between local session types and finite state machines to generate session types APIs for mainstream programming languages. [22, 23, 44, 40, 6] are not completely static: they check linearity dynamically. `MultiCrusty` can check linearity using the built-in affinity type checking from Rust. [43, 40, 6] do not enforce exhaustive handling of input types; and [22, 23, 32] rely on runtime checks to correctly handle branching.

[38, 54] provide static checking using the call-back style API generation. `MultiCrusty` uses a decomposition of AMPST to BST; in [44], MPST in Scala is implemented combining binary channels on the top of the existing BST library from [45]. Unlike `MultiCrusty`, [44] lacks static linearity check and uses a continuation-passing style translation from MPST into linear types. [32] implements static type-checking of communication protocols by linking Java classes and their respective typestate definitions generated from Scribble. Objects declaring a typestate should be used linearly, but a linear usage of channels is not statically enforced.

All above implementations generate multiparty APIs from protocols. To our knowledge, [26] is the only type-level embedding of classic multiparty channels in a mainstream language, OCaml. However, the library heavily relies on OCaml-specific parametric polymorphism for variant types to ensure type-safety. Their formalism lacks linear types and deadlock-freedom is not formalised nor proved. In addition, this implementation uses a non-trivial, complicated encoding of polymorphic variant types and lenses, while `MultiCrusty` uses the built-in affine type system in Rust.

The work most closely related to ours is [19] which implements handling of dynamic environments by MPST with explicit connections from [23], where actors can dynamically connect and disconnect. It relies on the actor-like research language, Ensemble; and generates endpoint code from Scribble. Their core calculus includes a syntax of the **try** L **catch** M construction where M is evaluated if L raises an exception. The type system follows [51], and is not as expressive as the previous paper on binary exception handling [16] that extends the richer type system of GV [37, 36]. Due to this limitation of their base typing system, and since their main focus is *adaptation*, there are several differences from AMPST, listed below: (1) they do not model general failure of multiple (interleaved) session endpoints (such as failures of *selection* and *branching* constructs as shown in rules [C-Sel], [C-Br]); (2) their **`try-catch`** scope (handler) is limited to a single action unlike AMPST and [16] where its scope

can be an arbitrary process P, participants and session endpoints ([R-Cat]); (3) they do not model any Rust specific ?-options where an arbitrary process P can self-fail ([T-try], [C-?Sel]); and (4) their kill process is weaker than ours (it is point-to-point, it does not broadcast the failure notification to the same session).

As a consequence, their progress result ([16, Theorem 18]) is weaker than our theorems since their configuration can be stuck with an exception process that contains **raise**, while our termination theorem (Theorem 3.22) guarantees that there always exists a path such that the process will move or terminate as **0**, *cleaning up* all intermediate processes which interact non-deterministically. More precisely, in [16, Theorem 18], a cancellation in a session is propagated, but **raise** blocks a reduction when the actor is not involved in a session, and its behaviour is also **stop**, meaning it is terminated. Otherwise, the actor will leave the session and restart. In contrast, `MultiCrusty` ensures the strong progress properties by construction (see § 2). We also implemented interleaved sessions (as shown in [33]), where one participant is involved in two different protocols at the same time.

As part of future work, we would like to develop recovery strategies based on causal analysis, along the lines of [41]. In addition, it would be interesting to verify role-parametric session types following [6] in an affine setting. Finally, we plan to study polymorphic meshed channels with different delivery guarantees such as TCP and UDP.

References

1 Davide Ancona, Viviana Bono, and Mario Bravetti. *Behavioral Types in Programming Languages*. Number 2-3 in Foundations and Trends in Programming Languages. Now Publishers Inc., Hanover, MA, USA, 2016. doi:10.1561/2500000031.

2 Stephanie Balzer and Frank Pfenning. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110281.

3 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest Deadlock-Freedom for Shared Session Types. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639, Cham, 2019. Springer. doi:10.1007/978-3-030-17184-1_22.

4 Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.

5 Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 222–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-15375-4_16.

6 David Castro, Raymond Hu, SungShik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. Place: New York, NY, USA Publisher: Association for Computing Machinery. doi:10.1145/3290342.

7 Ruofei Chen and Stephanie Balzer. Ferrite: A Judgmental Embedding of Session Types in Rust. *CoRR*, abs/2009.13619, 2020. arXiv:2009.13619.

8 Tokio Contributors. Crate: Tokio, 2021. Last accessed: July 2021. URL: https://crates.io/crates/tokio.

9 Wikipedia Contributors. Wikipedia: Flow (psychology), 2021. Last accessed: July 2021. URL: https://en.wikipedia.org/wiki/Flow_(psychology).

10 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016. doi:10.1017/S0960129514000188.

11 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693 of *PPoPP '22*, pages 261–246. ACM, 2022. doi:10.1145/3503221.3508404.

12 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *FMSD*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

13 Rust Developers. Rust: attempt-catch macro, 2018. Last accessed: July 2021. URL: https://play.integer32.com/?version=stable&mode=debug&edition=2018&gist=95979b17196adbc203c4f563e00d384b.

14 José Duarte and António Ravara. *Retrofitting Typestates into Rust*, pages 83–91. Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3475061.3475082.

15 Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Technical Report RFC7230, RFC Editor, June 2014. doi:10.17487/rfc7230.

16 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types Without Tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, January 2019. Place: New York, NY, USA Publisher: ACM. doi:10.1145/3290341.

17 Simon Gay and António Ravara. Behavioural Types: from Theory to Tools. In *Behavioural Types: from Theory to Tools*, Automation, Control and Robotics, pages 1–412. Rivers publishers, Alsbjergvej 10, 9260 Gistrup, Denmark, 2017. doi:10.13052/rp-9788793519817.

18 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434297.

19 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, page 30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2021.12.

20 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, ESOP '98, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/BFb0053567.

21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *SIGPLAN Not.*, 43(1):273–284, January 2008. doi:10.1145/1328897.1328472.

22 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering*, volume 9633, pages 401–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49665-724.

23 Raymond Hu and Nobuko Yoshida. Explicit Connection Actions in Multiparty Session Types. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, volume 10202, pages 116–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-662-54494-57.

24 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70592-5_22.

25 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.

26 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty Session Programming With Global Protocol Combinators. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.9.

27 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2808098.2808100.

28 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and Blame Assignment for Higher-Order Session Types. *SIGPLAN Not.*, 51(1):582–594, January 2016. doi:10.1145/2914770.2837662.

29 Aparicio Jorge. Crate: Criterion, 2021. Last accessed: July 2021. URL: https://crates.io/crates/criterion.

30 Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science*, 304:48–60, September 2019. doi:10.4204/eptcs.304.4.

31 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking Protocols with Mungo and stmungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 146–159, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2967973.2968595.

32 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and stmungo: A session type toolchain for Java. *Science of Computer Programming*, 155:52–75, April 2018. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016. doi:10.1016/j.scico.2017.10.006.

33 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay safe under panic: Affine rust programming with multiparty session types, 2022. doi:10.48550/ARXIV.2204.13464.

34 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From Communicating Machines to Graphical Choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 221–232, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676964.

35 Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117, Cham, 2019. Springer. doi:10.1007/978-3-030-25540-4_6.

36 Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In Jan Vitek, editor, *Programming Languages and Systems*, pages 560–584, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46669-8_23.

37 Sam Lindley and J. Garrett Morris. Talking Bananas: Structural Recursion for Session Types. *SIGPLAN Not.*, 51(9):434–447, September 2016. doi:10.1145/3022670.2951921.

38 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Generating Interactive WebSocket Applications in TypeScript. *Electronic Proceedings in Theoretical Computer Science*, 314:12–22, April 2020. doi:10.4204/EPTCS.314.2.

39 Dimitris Mostrous and Vasco T. Vasconcelos. Affine Sessions. *Logical Methods in Computer Science ; Volume 14*, 8459:Issue 4 ; 18605974, 2018. Medium: PDF Publisher: Episciences.org. doi:10.23638/LMCS-14(4:14)2018.

40 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A Session Type Provider: Compile-Time API Generation of Distributed Protocols with Refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 128–138, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3178372.3179495.

41 Rumyana Neykova and Nobuko Yoshida. Let It Recover: Multiparty Protocol-Induced Recovery. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 98–108, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3033019.3033031.

42 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local Verification of Global Protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *LNCS*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40787-1_25.

43 Nicholas Ng, Jose Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by Default. In Björn Franke, editor, *Compiler Construction*, pages 212–232, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46663-6_11.

44 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–LeibnizZentrum fuer Informatik. ISSN: 1868-8969. doi:10.4230/LIPIcs.ECOOP.2017.24.

45 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:28, Dagstuhl, Germany, 2016. Schloss Dagstuhl–LeibnizZentrum fuer Informatik. ISSN: 1868-8969. doi:10.4230/LIPIcs.ECOOP.2016.21.

46 Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290343.

47 Authors Scribble. Scribble home page, 2021. URL: http://www.scribble.org.

48 McArthur Sean. Crate: Hyper, 2021. Last accessed: July 2021. URL: https://crates.io/crates/hyper.

49 Company StackOverflow. Stackoverflow: 2020 Developer Survey, 2020. Last accessed: July 2021. URL: https://insights.stackoverflow.com/survey/2020.

50 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 865–878, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304069.

51 Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1):64–87, 2006. doi:10.1016/j.tcs.2006.06.028.

52 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, October 2021. doi:10.1145/3485501.

53 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, pages 22–41, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-05119-2_3.

54 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically Verified Refinements for Multiparty Protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428216.

How to Take the Inverse of a Type

Daniel Marshall
School of Computing, University of Kent, Canterbury, UK

Dominic Orchard
School of Computing, University of Kent, Canterbury, UK
Department of Computer Science and Technology, University of Cambridge, UK

Abstract

In functional programming, regular types are a subset of algebraic data types formed from products and sums with their respective units. One can view regular types as forming a commutative semiring but where the usual axioms are isomorphisms rather than equalities. In this pearl, we show that regular types in a *linear* setting permit a useful notion of *multiplicative inverse*, allowing us to "divide" one type by another. Our adventure begins with an exploration of the properties and applications of this construction, visiting various topics from the literature including program calculation, Laurent polynomials, and derivatives of data types. Examples are given throughout using Haskell's linear types extension to demonstrate the ideas. We then step through the looking glass to discover what might be possible in richer settings; the functional language Granule offers linear functions that incorporate local side effects, which allow us to demonstrate further algebraic structure. Lastly, we discuss whether dualities in linear logic might permit the related notion of an *additive inverse*.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases linear types, regular types, algebra of programming, derivatives

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.5

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.1`

Funding This work is supported by an EPSRC Doctoral Training Award (Marshall) and EPSRC grant EP/T013516/1 (Verifying Resource-like Data Use in Programs via Types).

Acknowledgements Thanks to Nicolas Wu and Harley Eades III for their valuable comments and discussion on earlier drafts, and also to the anonymous reviewers for their helpful feedback.

1 Prologue: Consuming with Inverses

Algebraic data types are the bread-and-butter of both the theory and practice of functional programming. The algebraic view gives rise to vast possibilities for manipulating types, and for "calculating" programs from their type structure in the Bird-Meertens tradition [7, 8, 37].

Regular types are a subset of algebraic types formed from products $\times$, sums $+$, unit 1 and empty types 0, and fixed points, giving rise to polynomial type expressions [37, 40] and an algebraic structure akin to a commutative semiring. The multiplicative part is by products and the unit type, and the additive part by sums and the empty type. However, the semiring laws are relaxed to isomorphisms, e.g., $a \times (b \times c) \cong (a \times b) \times c$ is witnessed by a bijection between the two ways of associating a triple expressed as pairs. The cardinality operation $|-|$ (mapping a type to its size) is then a semiring homomorphism (a functor) from the structure of types to natural numbers, e.g., $|a \times b| = |a||b|$. This provides a useful technique for understanding when different type expressions are isomorphic by checking if their cardinalities are equal, a fact leveraged by many a student for decades. In category theory, we can model regular types as a *(commutative) semiring category* (or *rig category* [11]) or a *(symmetric) bimonoidal category* [26], with semiring rules as natural isomorphisms. Either way, a rose by any other name is still as sweet.

© Daniel Marshall and Dominic Orchard;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 5; pp. 5:1–5:27

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Given the rich algebraic analogues and models for types, one may then (perhaps idly) wonder: if product types are multiplicative, *is there a notion of multiplicative inverse for types which lets us divide one type by another*? We show that this question has a rather neat and simple answer for *linear types*:

▶ **Definition 1.** The multiplicative inverse of type τ is the type of functions *consuming* τ:

$$\tau^{-1} \triangleq \tau \multimap 1$$

where $\multimap$ is the linear function type, of functions which use their argument exactly once.

Linear types are an ideal setting for capturing the idea of *consumption* since linear types treat values as resources which must be used exactly once – they can never be discarded (no weakening) or duplicated or shared (no contraction) [20, 51, 55]. Therefore, a function $\tau \multimap 1$ must consume its argument rather than simply returning a value of the unit type.

Linear regular types have products $\otimes$ ("multiplicative conjunction") where each component of the pair is used exactly once and sums $\oplus$ ("additive disjunction") which behave like normal sum types, though whichever component we are given must of course be used linearly. Linear regular types again form a commutative semiring structure, with equalities as isomorphisms.

Given $\tau^{-1} \triangleq \tau \multimap 1$, we can immediately consult the standard meaning of multiplicative inverse with regards to its equational theory: a group extends a monoid $(X, \bullet, e)$ such that for every $x \in X$ there exists an *inverse* element denoted $x^{-1} \in X$, for which $x \bullet x^{-1} = e$ and $x^{-1} \bullet x = e$. If the use of the notation $-^{-1}$ and the terminology of inverses is warranted then we could reasonably expect $\tau^{-1} \otimes \tau \cong 1$. Given the above definition of inverse, then one direction of this isomorphism, from $\tau^{-1} \otimes \tau$ to 1, is inhabited via function application:

$$\lambda(u, x).\ u\ x\ :\ (\tau^{-1} \otimes \tau) \multimap 1 \tag{1}$$

where the first component of the pair consumes the second component. Similarly we can construct the symmetric version $(\tau \otimes \tau^{-1}) \multimap 1$ by first flipping the components of the pair.

These are the only inhabitants of their types: the multiplicative inverse of τ *must* consume the τ value in the other component of the pair due to the constraints of linearity. Indeed, starting from the goal of defining a multiplicative inverse for a linear type such that the property $\tau^{-1} \otimes \tau \multimap 1$ holds, by "currying" we must have a map from τ^{-1} to $\tau \multimap 1$. Therefore, $\tau \multimap 1$ is the most natural choice for an inverse to τ, as any other inverse would first need to be mapped to $\tau \multimap 1$ in order to consume the value of type τ.

We call equation (1) and its symmetric counterpart *lax inverse laws* following category theory terminology: *strict* structures have equalities, *strong* have isomorphisms, and *lax* have only morphisms in one direction. The rest of the commutative semiring structure of linear regular types is "strong" as associativity, commutativity, etc. are isomorphisms. However, for inverses, the opposite direction from 1 to $\tau^{-1} \otimes \tau$ does not exist in general: for any τ we cannot necessarily form a pair $\tau^{-1} \otimes \tau$ as we do not have an algorithm to construct an arbitrary τ value nor its consumer. Even if we can inhabit $1 \multimap (\tau^{-1} \otimes \tau)$ for a particular τ (e.g., if there is a default value for a type and a standard way of consuming its values) this only forms an isomorphism $\tau^{-1} \otimes \tau \cong 1$ in the limited setting of 1-element types. The crux is that division *loses* information: $\tau^{-1} \otimes \tau \multimap 1$ consumes knowledge of the original τ value.

In a non-linear ("Cartesian") setting, we could still define inverses in a similar way as $\tau^{-1} \triangleq \tau \to 1$, and the lax inverse law $\tau^{-1} \times \tau \to 1$ would be similarly inhabited by function application. However, non-linearity makes this definition weaker and less natural, as $((\tau \to 1) \times \tau) \to 1$ is also inhabited by the function $\lambda(u, x).1$ which throws away both of its arguments and returns the unit value. We could just as easily construct this term of

type $(\tau^{-1} \times \tau) \to 1$ regardless of the definition we chose for τ^{-1}. It is only by working in the context of linear types that our notion of inverse is given meaning, as a function $\tau \otimes \tau^{-1} \multimap 1$ must use both the term τ and its inverse, for which $\tau^{-1} \triangleq \tau \multimap 1$ is the natural fit.

This definition of τ^{-1} lets us view types as an algebraic structure which is *almost* a semifield; a semifield resembles a semiring except that every nonzero element has a multiplicative inverse. The terminology of *skewness* can also be applied here; for example, a *skew monoid* is one in which the associativity and unit properties are morphisms rather than isomorphisms or equalities [48, 50]. Thus our construction could be described as a "multiplicative skew inverse". Going forwards we use just "inverse" for brevity.

Roadmap

In this pearl, we explore various applications and consequences of this idea. We begin by programming with inverses in Haskell via the linear types extension of the Glasgow Haskell Compiler[1] (based on the work of Bernardy et al. [6]), and proceed to consider equations arising from functions over inverses and other algebraic implications.

One interesting result we uncover is that whilst regular types yield polynomial type expressions, inverse types yield an analogue of the mathematical generalisation of *Laurent polynomials* (Section 4); these differ from ordinary polynomials in that they can have terms of negative degree, providing a more general notion of exponent for regular types. But inverses turn out to have applications beyond the merely theoretical; we show inverses allow the notion of derivatives for regular types (à la McBride [37]) to be generalized, providing the ability to take the derivative of a type *with respect to another type* (Section 5). This yields a way to generate data types with n-holes (holes of n contiguous elements) which we apply to the common programming idiom of stencil computations.

In the second half, we consider possibilities for developing the algebraic structure of inverse types further, by working in richer and more expressive settings. It turns out that the multiplicative inverse becomes an involution (Section 6) if we can express *sequentially-realizable functions* [32] which carry out local side effects that are not observable externally. We demonstrate this using linear session channels à la Lindley and Morris [30] which allow inverses to do more computationally. We show examples in the modern functional language Granule which has linear types at its core [42]. This involution also happens to yield a construction akin to the familiar continuation monad, which we briefly discuss.

Lastly, we show it is also possible to define an additive inverse (Section 7). However, this requires working in a different setting where products are given by linear logic's & rather than $\otimes$ and similarly sums are $\parr$ rather than $\oplus$. Thus, while we can develop the theory for each kind of inverse separately, there is not yet any type theory where the two can coexist.

A side aim of this pearl is to popularise the increasing abilities of modern functional languages to express linear types. For those unfamiliar with linear types and wishing to go beyond the intuitions here, Appendix A gives some standard typing rules and syntax. We also provide an artifact[2] including all code examples given throughout in both Haskell and Granule, to aid with understanding and allow for further experimentation.

---

[1] Available as of GHC 9.0.1, `https://www.haskell.org/ghc/download_ghc_9_0_1.html`, released Feb 2021.
[2] `https://doi.org/10.5281/zenodo.6275280`

2 Programming with Inverses

We stand at an exciting juncture for our community. Finally, more than 30 years after their conception in logic [20], linear types are starting to gain a foothold in mainstream functional programming languages. One such language is Haskell, via GHC's linear types extension [6] which uses a graded type system [42] based on annotating function types `a %r -> b` with their "multiplicity" `r` (which can also be understood as a *coeffect*, or *consumption effect* [43]), that describes how many times the argument is used. In Haskell, this can either be `1` representing *linear* behaviour or `Many` representing *unrestricted* behaviour, including the possibility of 0 uses. We can thus describe inverses and a curried version of the lax inverse law as follows:[3]

```
1  type Inverse a = a %1 -> ()      -- recall () is the unit type 1 of Haskell
2
3  divide :: a %1 -> Inverse a %1 -> ()
4  divide x u = u x
```

The naming of `divide` is to evoke the usual intuition associated with groups where $a/b = a \bullet b^{-1}$ and since this function "actions" the consumption of the first input by the second.

There are other linearly-typed languages in which one could also readily apply our notion of inverses, e.g., ATS [47], Alms [49], and Quill [39]. Through some translation we can also represent inverses in languages with more expressive graded type systems, such as Granule [42] and Idris 2 [10], that can describe linearity as well as other flavours of resourceful data. We focus on Haskell for now, but the ideas are the same no matter the language.

In the concrete setting of an actual language, we can now give an example inhabitant of an inverse type. These are typically defined by some pattern matching over all the possible inputs, where the act of pattern matching on the incoming value consumes the input as it inspect its value. For example, an inverse to Haskell's boolean type is given by:

```
1  boolDrop :: Inverse Bool
2  boolDrop True  = ()
3  boolDrop False = ()
```

The `linear-base` library for Haskell provides a type class for those types which are "consumable": inhabitants of the inverse of type `a`. The instance of `Consumable` for the boolean type is the `boolDrop` function defined explicitly above.

```
1  class Consumable a where
2     consume :: a %1-> ()
3
4  instance Consumable Bool where
5    consume True  = ()
6    consume False = ()
```

Various built-in types like `Int` have a "linearly unsafe" implementation which simply drops the argument rather than, say, consuming a machine integer by matching on the `0` case and otherwise recursively consuming the integer decremented by `1`, which would be safe but slow! This explicit weakening operation can also be algorithmically generated from a regular type, following a generic deriving mechanism [23].

[3] Note that we first need to enable the linear types extension, by using the pragma `{-# LANGUAGE LinearTypes #-}`; this will be left implicit in all of the snippets of Haskell throughout the pearl.

A key aspect of this typing discipline is that we do not want certain types to be consumable without side effects; for example, file handles, sockets, channels, or any other piece of data which acts as a proxy for a resource for which there exists some protocol of interaction. In Section 6, we see more interesting inhabitants of inverse types in a more expressive setting.

We can consider algebraic properties of inverses and understand them through the lens of linear regular types using this definition, while bearing in mind that our inverses are lax, and so the properties will hold only in one direction. For example, consider the following property, which is a simple application of the distributivity of multiplication over addition.

$$\begin{aligned}(\tau \oplus 1) \otimes \tau^{-1} &\cong ((\tau \otimes \tau^{-1}) \oplus \tau^{-1}) \\ &\multimap 1 \oplus \tau^{-1}\end{aligned} \tag{2}$$

We can understand $\tau \oplus 1$ as the linear version of the traditional Haskell `Maybe` data type (called *option* in ML), and thus recover the following function definition in Haskell corresponding to the above (in)equation, giving us a way to distribute an inverse into a `Maybe` value.

```
1 maybeNeg :: (Maybe a) %1 -> Inverse a %1 -> Maybe (Inverse a)
2 maybeNeg Nothing  u = Just u
3 maybeNeg (Just n) u = letUnit (divide n u) Nothing
4
5 letUnit :: () %1 -> a %1 -> a -- Abstracts ‘let () = t1 in t2‘ - needed since
6 letUnit () x = x               -- let bindings are currently always non-linear.
```

In the second case of `maybeNeg`, we cannot simply return `Nothing` since `u` and `n` are linearly typed; we must first apply `u` to `n` (via `divide`) to consume both values. We then want `let () = divide n u in Nothing`, but linear let-bindings are not yet implemented (as of GHC 9.2.2, released in March 2022), so we abstract this pattern as the function `letUnit` instead.

3 Calculating with Inverses

Regular types come equipped with various equations governing their operations which can be used for reasoning about functional programs [18] and even deriving implementations starting from equational specifications (the *Bird-Meertens* formalism) [7, 8, 19]. We consider here analogous equations for calculating with inverses. We explore these equations from the perspective of the linear λ-calculus with regular types, illustrating some points using Haskell for convenience. One can freely translate between the two.

In a linear types setting, many of the usual equations governing products are not available to us, because the "tupling" that combines regular functions $f : A \to B$ and $g : A \to C$ into $\langle f, g\rangle : A \to (B \times C)$ violates linearity by copying a value of type A, and projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ violate linearity by discarding one component of a pair. We can however work with $\otimes$ as a bifunctor (which lifts $f : A \multimap B$ and $g : C \multimap D$ to $f \otimes g : A \otimes C \multimap B \otimes D$), and cotupling $[h, k] : A \oplus B \multimap C$ (for $h : A \multimap C$ and $k : B \multimap C$) is still available. Thus we have equations for (bi)functoriality of $\otimes$ and $\oplus$:

$$\begin{aligned} id \otimes id &= id & id \oplus id &= id \\ (f \otimes g) \circ (h \otimes k) &= (f \circ h) \otimes (g \circ k) & (f \oplus g) \circ (h \oplus k) &= (f \circ h) \oplus (g \circ k)\end{aligned}$$

and equations interacting cotupling, injections and the $\oplus$ bifunctor, e.g., to name a few:

$$[f, g] \circ \mathsf{inl} = f \qquad [f, g] \circ \mathsf{inr} = g \qquad [h \circ \mathsf{inl}, h \circ \mathsf{inr}] = h$$

For brevity we elide the rest as they are not the object here. Appendix A.1 gives the remaining equations (which are the subset of those from Gibbons [18] that are permitted in a linear setting). There are various other equations arising from the isomorphisms of regular types (Section 1), e.g., for the isomorphisms witnessing associativity with $\alpha : (A \otimes B) \otimes C \multimap A \otimes (B \otimes C)$ and α_i is its converse, then we have equations $\alpha \circ \alpha_i = \alpha_i \circ \alpha = id$.

Inverse as a functor

A few equations arise from the simple fact that multiplicative inverse is a contravariant functor, and thus we have a contravariant "map" function via function composition:

```
comap :: (b %1 -> a) %1 -> Inverse a %1 -> Inverse b
comap f g = \x -> g (f x)
```

A functor's action on a morphism is commonly written using the same symbol as its action on objects (types), just as seen above for $\otimes$ and $\oplus$. However, writing `comap` applied to f as f^{-1} gives the wrong impression: we are not representing the inverse of a function, but rather lifting a function to work on inverses. We therefore write $f^{\ominus 1}$ for `comap` f to avoid confusion.[4]

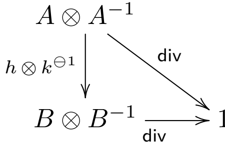
We therefore have the functoriality equations for inverses:

$$id^{\ominus 1} = id \qquad g^{\ominus 1} \circ f^{\ominus 1} = (f \circ g)^{\ominus 1}$$

Let $\mathsf{div} : A \otimes A^{-1} \multimap 1$ be the lax inverse law as a function in the linear λ-calculus (the uncurried form of the function `divide :: a -> Inverse a -> ()` we defined for Haskell earlier). Given two functions $h : A \multimap B, k : B \multimap A$, we then have the following naturality property:

$$\mathsf{div} \circ (h \otimes k^{\ominus 1}) = \mathsf{div}$$

which can be seen more clearly in a diagram as:



i.e., transforming a value and its inverse prior to consumption is the same as us just consuming the original value. This property follows from the definitions. We revisit this law in Section 6 once we introduce consuming functions that can have some (safe-by-linearity) side effect.

Monoidal structure of inverses

The inverse (contravariant) functor also has additional monoidal functor structure, which we can write in Haskell simply as:

```
munit :: () %1 -> Inverse ()
munit () = (\() -> ())

mmult :: (Inverse a) %1 -> (Inverse b) %1 -> Inverse (a, b)
mmult f g = \(a, b) -> letUnit (f a) (g b)
```

[4] We considered using the notation $\tau^{\ominus 1}$ for types as well, but thought it was too ugly to put everywhere.

i.e., `munit` consumes a unit value by pattern matching then returning unit (the standard polymorphic identity function would have worked equally well), and `mmult` returns a function that consumes a pair by using `f` to consume the first component then `g` to consume the second. The usual axioms of a (lax) monoidal functor (associativity and unit laws) [34] hold, interacting with the monoidal structure of $\otimes$; we detail these axioms in Appendix A.1.

This monoidal functor structure on inverses gives us the simple idea that we can combine multiple inverses into a composite inverse; in other words, a pair of consumers can be turned into a consumer of pairs. This satisfies the following equation:

$$\rho \circ (\mathsf{div} \otimes \mathsf{div}) = \mathsf{div} \circ (id \otimes \texttt{mmult}) \circ interchange$$

where $\rho : 1 \otimes 1 \multimap 1$ collapses units and $interchange : (A \otimes B) \otimes (C \otimes D) \multimap (A \otimes C) \otimes (B \otimes D)$ is derived from associativity and commutativity isomorphisms. This rule can be seen more clearly as a diagram:

$$\begin{array}{ccccc}
(A \otimes A^{-1}) \otimes (B \otimes B)^{-1} & & \xrightarrow{interchange} & & (A \otimes B) \otimes (A^{-1} \otimes B^{-1}) \\
\downarrow{\scriptstyle \mathsf{div} \otimes \mathsf{div}} & & & & \downarrow{\scriptstyle id \otimes \texttt{mmult}} \\
1 \otimes 1 & \xrightarrow[\rho]{} & 1 & \xleftarrow[\mathsf{div}]{} & (A \otimes B) \otimes (A \otimes B)^{-1}
\end{array}$$

i.e., we can perform two inverse eliminations or we can rearrange, combine the inverses into one, and then apply a single elimination.

The idea of combining products of inverses together leads naturally to notions of *exponentiation*, but with negative exponents, which we explore next.

4 Exponentiation with Inverses

In the standard semiring of (linear) regular types discussed in Section 1, the type constructors 0, 1, $\otimes$ and $\oplus$ generate polynomials over some type meta-variable where terms with exponents τ^n are represented by the n-wise product of τ where $\tau^0 = 1$. For $n \geq 0$, this gives us the usual positive exponent laws up to isomorphism (using associativity and commutativity):

$$\forall a, b.\, (a \geq 0 \land b \geq 0) \quad \tau^a \otimes \tau^b \cong \tau^{a+b} \qquad (\exp_{+})$$

$$\forall a.\, (a \geq 0) \quad \sigma^a \otimes \tau^a \cong (\sigma \otimes \tau)^a \qquad (\exp_{\sigma})$$

Introducing the multiplicative inverse allows us to generalise these to negative exponents, and thus to generate *Laurent polynomials* over types, which differ from ordinary polynomials in that they can have terms of negative degree [28].

We define exponentiation over a type τ for negative exponents as

$$\forall a.(a \geq 0) \qquad \tau^{-a} \triangleq (\tau^{-1})^a$$

For example, $\tau^{-2} = (\tau^{-1})^2 = \tau^{-1} \otimes \tau^{-1}$ capturing a pair of inverses.

It is clear that the first exponential law (equation $\exp_{+}$) generalises in the case that both coefficients are negative, since we define τ^{-n} as a product of n inverses τ^{-1} in much the same way that τ^n represents a product of n values of type τ, i.e.,

$$\forall a, b.\, (a \geq 0 \land b \geq 0) \quad \tau^{-a} \otimes \tau^{-b} \cong \tau^{-(a+b)} \qquad (\exp_{-})$$

This is an isomorphism as it amounts to re-associating products.

In the case that just one coefficient is negative, our notion of inverse satisfies a generalisation of the exponentiation law as a lax property:

$$\forall a, b. (a \geq 0 \wedge b \geq 0) \quad \tau^a \otimes \tau^{-b} \multimap \tau^{a-b} \qquad (\exp_{\pm})$$

The lax inverse law $\tau \otimes \tau^{-1} \multimap 1$ is then a specialisation of the above with $a = b = 1$ since $\tau^0 = 1$. As another example, consider $a = 5$ and $b = 3$; then we can eliminate/consume three elements of a quintuple to get pairs ($a - b = 2$). This raises an interesting combinatorial question about the number of functions inhabiting this type (witnessing this lax property).

▶ **Proposition 1.** *It turns out that the number of possible witnesses of the lax inverse law* $\tau^a \otimes \tau^{-b} \multimap \tau^{a-b}$ *if* $a \geq 0$, $b \geq 0$ *and* $a \geq b$ *is simply* $\binom{a}{b} \times (a-b)! = \frac{a!}{b!}$.

The intuition for this is that if we have some number a of values and some number b of inverses, and we have more values than inverses, then we must apply every single inverse to a value, and the only choice we can make is which values we are going to consume. Combinatorially there are $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ ways to choose b of the a elements, leaving $(a - b)$ elements remaining in the end which we can permute in any order (hence the $(a - b)!$ factor which cancels out). If $b \geq a$, then we must instead consume every value, and so conversely we are simply choosing which a inverses we will use to do this, giving a result of $\frac{b!}{a!}$.

Thus far in the paper we have examined the linear λ-calculus and linear types in Haskell, where inverses have no significant computational content beyond consuming a linear value. However, later in Section 6 we will work in a setting with inverses that incorporate local side effects. Notice that in such a setting the order in which we apply inverses may be important. Thus, we also consider the result we obtain when taking this into consideration.

▶ **Proposition 2.** *If reorderings are considered distinct, then the number of possible witnesses of the lax exponentiation law* $\tau^a \otimes \tau^{-b} \multimap \tau^{a-b}$ *if* $a \geq 0$, $b \geq 0$ *and* $a \geq b$ *is a* factorial *($a!$).*

This is derived by $P(a, b) \times (a - b)!$, i.e. $\dfrac{a!}{(a-b)!} \times (a - b)! = a!$ since again we can "consume" elements via their inverses in any order, giving the number of permutations $P(a, b)$ of length b taken from a, and there are $(a - b)$ remaining elements left afterwards to permute. Here we run through the proof in full, as it is instructive about how to inhabit the law in either case.

Proof. We prove that the number of possible witnesses of the lax exponentiation law is as given above by induction on b (and recall we assume $a \geq b$).

- ($b = 0$) For the base case, we are then considering $|\tau^a \otimes \tau^0 \multimap \tau^{a-0}| = |\tau^a \multimap \tau^a|$ since $\tau^0 = 1$. The possible inhabitants of $\tau^a \multimap \tau^a$ are then just permutations of the a-wise product of τ and so $|\tau^a \multimap \tau^a| = a!$, giving the result here.
- ($b = 1$) We next consider another case where $b = 1$ since this is instructive (though not necessary). Here we thus need to find how many ways there are to inhabit $\tau^a \otimes \tau^{-1} \multimap \tau^{a-1}$, i.e., how many ways to "cut out" one τ from an a-wise product of τ.
  We can construct a many terms as witnesses for this type by picking any one of the τ values to consume with the inverse τ^{-1}:

$$\begin{aligned}
\lambda((a_1, a_2, \ldots, a_n), u) &= \mathbf{let}\ () = u\ a_1\ \mathbf{in}\ (a_2, \ldots, a_n) &&: (\tau^a \otimes (\tau \multimap 1)) \multimap \tau^{a-1} \\
\lambda((a_1, a_2, \ldots, a_n), u) &= \mathbf{let}\ () = u\ a_2\ \mathbf{in}\ (a_1, a_3, \ldots, a_n) &&: (\tau^a \otimes (\tau \multimap 1)) \multimap \tau^{a-1} \\
&\ldots \\
\lambda((a_1, a_2, \ldots, a_n), u) &= \mathbf{let}\ () = u\ a_n\ \mathbf{in}\ (a_1, a_2, \ldots, a_{n-1}) &&: (\tau^a \otimes (\tau \multimap 1)) \multimap \tau^{a-1}
\end{aligned}$$

Whichever witness we choose, one inverse has been applied, so $a-1$ of the original elements remain. These can be permuted in any order, giving a total of $a \times (a-1)! = a!$ witnesses.

- $(b = n+1)$ (Inductive step) We need to show that $|\tau^a \otimes \tau^{-(n+1)} \multimap \tau^{a-(n+1)}| = a!$. We know $\tau^{-(n+1)} \cong \tau^{-1} \otimes \tau^{-n}$ as we can simply "split off" one of the $n+1$ inverses from the remaining product of n inverses. Similarly to the base case $(b = 0)$, we first apply this inverse to any of the elements of τ^a; there are a possible choices here. This leaves $a-1$ of the original elements remaining, and n inverses. Therefore we can reason as follows:

$$\begin{array}{lll}
 & |\tau^a \otimes \tau^{-(n+1)} \multimap \tau^{a-(n+1)}| & \\
= & |(\tau^a \otimes \tau^{-1}) \otimes \tau^{-n} \multimap \tau^{a-(n+1)}| & \text{(split off one inverse, as above)} \\
= & a \times |\tau^{a-1} \otimes \tau^{-n} \multimap \tau^{(a-1)-n}| & \text{(a ways to apply one inverse)} \\
= & a \times (a-1)! & \text{(induction with $a-1$)} \\
= & a! & \square
\end{array}$$

Note, $a \geq (n+1)$ implies $(a-1) \geq n$, which is needed to inductively apply the proposition. Another way to see this intuitively is as follows. For $|\tau^a \otimes \tau^{-b} \multimap \tau^{a-b}|$ (with $a \geq b$), in order to find a witness we simply need to arrange the a elements in any order, such that the first b are assigned to the b inverses and the remaining $a-b$ are left unused. There are $a!$ ways to arrange the a elements, giving the result. Again, in the case that $b \geq a$ we just consider the b inverses instead, assigning inverses to elements, giving a total of $b!$ orderings. ◀

After that *divertimento* into the inhabitants of the negative exponentiation law $(\exp_{\pm})$ and its cardinality, we consider the second exponentiation law (sometimes called *power of a product*), which for regular types is the following isomorphism, by commutativity and associativity:

$$\forall a. \, (a \geq 0) \quad \sigma^a \otimes \tau^a \cong (\sigma \otimes \tau)^a \qquad (\exp_\sigma)$$

For a version of this law with negative exponents, a special case with $a = -1$ is already provided by the monoidal functor structure of Section 3 with $\mathsf{mmult} : \sigma^{-1} \otimes \tau^{-1} \multimap (\sigma \otimes \tau)^{-1}$ which combines inverses. Composing this with the double-negative-exponents law exp_{-} : $\tau^{-a} \otimes \tau^{-b} \cong \tau^{-(a+b)}$ yields the generalisation to all negative exponents:

$$\forall a. \, (a \geq 0) \quad \sigma^{-a} \otimes \tau^{-a} \multimap (\sigma \otimes \tau)^{-a} \qquad (\exp_{-\sigma})$$

For example, if $a = 2$ then this is derived as:

$$\sigma^{-2} \otimes \tau^{-2} \overset{\alpha+\gamma}{\cong} (\sigma^{-1} \otimes \tau^{-1}) \otimes (\sigma^{-1} \otimes \tau^{-1}) \overset{\mathsf{mmult} \otimes \mathsf{mmult}}{\multimap} (\sigma \otimes \tau)^{-1} \otimes (\sigma \otimes \tau)^{-1} \overset{\mathsf{exp}_{-}}{\cong} (\sigma \otimes \tau)^{-2}$$

where the isomorphism on the left applies associativity α and commutativity γ.

Overall, negative exponents generalise the "inverses as consumers" story. Given a product of two exponentiated types, one positive and one negative, then "actioning" the inverses involves consuming some number of elements of a product, leaving us with the remainder. This captures the notion of "projection", where some part of a type is thrown away and some part is retained. This pattern is relevant next, where having inverses and negative exponents for regular types allows us to define the derivative of a type *with respect to another type*.

5 Differentiating with Inverses

With our notion of multiplicative inverse in hand, we can apply other ideas from mathematics which rely on the presence of division.

First, let's recall the remarkable feature of regular types (with added type variables) that one can compute their *derivative* by applying the laws of Newton-Leibniz calculus [29]. This produces a companion data type of "one-hole contexts" for the original type, a beautiful idea due to McBride [37]. For example, for the parametric type α^4 (4-tuples with elements all of the same type) its derivative with respect to α is $4\alpha^3$, equivalent to $\alpha^3 + \alpha^3 + \alpha^3 + \alpha^3$. The intuition behind this is that there are four distinct ways in which you can take the original data type (4-tuples) and remove a single element (creating a hole), leaving the surrounding context (a triple of the three remaining elements). We can visualise these four possibilities (each of type α^3) as follows, where $-$ represents the "hole" and where $x, y, z, w : \alpha$:

$$(-, y, z, w) \qquad (x, -, z, w) \qquad (x, y, -, w) \qquad (x, y, z, -)$$

We write this derivative as $\partial_\alpha(\alpha^4)$, i.e., the partial derivative with respect to α keeping other variables constant (including recursion variables or other type parameters).

This approach can be applied to recursive regular types. For example, McBride's technique calculates $\partial_\alpha(\mathsf{List}\ \alpha) = \partial_\alpha(\mu X.1 + \alpha \times X) = (\mathsf{List}\ \alpha) \times (\mathsf{List}\ \alpha)$ representing the idea that a list with a single hole is equivalent to a pair of lists – the prefix of elements before the hole and the suffix of elements after the hole. The data type of list zippers (à la Huet [22]) is then given by $\alpha \times \partial_\alpha(\mathsf{List}\ \alpha)$ where we have a value α which "fills" the hole position, paired with its context. It is possible to extend this notion further to consider derivatives of general data types which act as containers [1] or even data which is in the process of being transformed from one type to another [38], but here we will concentrate on the simpler cases.

A data type of contexts with *two* holes can be obtained by repeated differentiation, i.e., $\partial_\alpha(\partial_\alpha(f(\alpha)))$, and thus we can compute contexts with n holes by taking the n-th derivative. Such holes are all independent; they can appear anywhere in the type. For example, $\partial_\alpha(\partial_\alpha(\mathsf{List}\ \alpha)) = (\mathsf{List}\ \alpha \times (\mathsf{List}\ \alpha \times \mathsf{List}\ \alpha)) + ((\mathsf{List}\ \alpha \times \mathsf{List}\ \alpha) \times \mathsf{List}\ \alpha)$, representing two ways of adding another hole to a list which already has one hole: either we have another hole in the left sublist or in the right sublist.

Though the derivatives above are based on standard regular types which have the structure of intuitionistic logic, linear regular types form a semiring in exactly the same syntactic manner. Thus, the notion of taking the derivative of a type applies equally well in the linear setting. We consider derivatives of linear regular types going forward, and show that inverses allow us to define what it means to take a derivative *with respect to another type*.

First, let's stay on the firm ground of real analysis. Recall from calculus that we can take the derivative of a function f with respect to another function g by the following method:

$$\partial_{g(\alpha)}(f(\alpha)) = \frac{\partial_\alpha(f(\alpha))}{\partial_\alpha(g(\alpha))} \tag{3}$$

For example, taking $f(\alpha) = \alpha^4$ and $g(\alpha) = \alpha^2$, then:

$$\partial_{(\alpha^2)}\alpha^4 = \frac{\partial_\alpha(\alpha^4)}{\partial_\alpha(\alpha^2)} = \frac{4\alpha^3}{2\alpha} = 2\alpha^2 \tag{4}$$

Giving this an interpretation in regular types, rather than $\mathbb{R}$, recall α^4 is a 4-tuple (a quadruple) and α^2 is a 2-tuple (a pair). Differentiating α^4 with respect to α^2 yields $2\alpha^2$ which is the data type capturing the two possible contexts obtained by removing a pair from the original type, leaving two elements in the remaining context. We call the pair removed here a *2-hole*, as it captures two contiguous (adjacent) holes. Note that this is different to the case, described earlier, of differentiating with respect to α twice; this gave two independent holes which did not necessarily have to be contiguous. The resulting type here $2\alpha^2 = \alpha^2 + \alpha^2$ can be interpreted as the type of 2-hole contexts, illustrated as:

$$(-_1, -_2, z, w) \qquad \textit{and} \qquad (x, y, -_1, -_2) \tag{5}$$

where $-_1$ and $-_2$ correspond to the two successive components of the 2-hole. We are not allowed to have the 2-hole splitting the remaining two elements like $(x, -_1, -_2, y)$; the data type of 2-hole contexts views the whole type in terms of pairs.

The derivative calculated in (4) was in $\mathbb{R}$ and *then* we interpreted the result as a type. This however does not generalise well. Consider the derivative of α^3 with respect to α^2:

$$\partial_{(\alpha^2)}\alpha^3 = \frac{\partial_\alpha(\alpha^3)}{\partial_\alpha(\alpha^2)} = \frac{3\alpha^2}{2\alpha} = \frac{3}{2}\alpha$$

We simplified the result following the axioms of fields here but are left with the unwieldy $\frac{3}{2}$ which we cannot meaningfully translate into the realm of types. Using our approach of multiplicative inverses instead yields a more generally applicable result for regular types.

▶ **Definition 2.** The derivative of a regular type with respect to another regular type is:

$$\partial_{g(\alpha)} f(\alpha) = \partial_\alpha f(\alpha) \otimes (\partial_\alpha g(\alpha))^{-1} \tag{6}$$

This construction yields the usual derivative of the numerator, paired with a consumer of the derivative of the denominator.

Returning to the example of taking the derivative of α^4 with respect to α^2 shown in equation (4), we instead apply the inverses approach of Definition 2 which yields:

$$\partial_{\alpha^2}(\alpha^4) = 4\alpha^3 \otimes (2\alpha)^{-1}$$

This is the type $4\alpha^3$ of 1-hole contexts for 4-tuples (the four possible triples resulting from removing one element) paired with an inverse which can be used to consume a further α value to create a 2-hole. This inverse consumes 2α values, i.e., $\alpha \oplus \alpha$, where the α value is tagged with an extra bit of information to explain to the inverse which element is being removed to create the 2-hole. We can see this as the four possible 1-holes with an inverse $\iota : (2\alpha)^{-1}$ which can be actioned to recover the 2-holes for 4-tuples as in equation (5):

$$(-, y, z, w) \otimes \iota \xrightarrow{\iota(\mathsf{inr}\ y)} (-, -, z, w) \xleftarrow{\iota(\mathsf{inl}\ x)} (x, -, z, w) \otimes \iota \qquad (x, y, -, w) \otimes \iota \xrightarrow{\iota(\mathsf{inr}\ w)} (x, y, -, -) \xleftarrow{\iota(\mathsf{inl}\ z)} (x, y, z, -) \otimes \iota \tag{7}$$

with the usual injections $\mathsf{inl} : A \multimap A \oplus B$ and $\mathsf{inr} : B \multimap A \oplus B$.

We put all this together in Haskell to define a notion of 1-hole contexts for 4-tuples (`QuadContexts` below), which we pair with consumers to create 2-holes (`QuadTwoContexts` below):

```
1  -- Represents 4a^3 (the four possible ways to remove one element from a 4-tuple).
2  data QuadContexts a =
3      Mk1 a a a -- context: -, y, z, w
4    | Mk2 a a a -- context: x, -, z, w
5    | Mk3 a a a -- context: x, y, -, w
6    | Mk4 a a a -- context: x, y, z, -
7
8  data QuadTwoContexts a = Mk (QuadContexts a) (Inverse (Either a a))
```

We can then use the inverses, as in the above illustration, to map from a 2-hole and a context back into the original α^4 type, implementing the illustration of equation (7):

```
fromContext :: (a, a) %1 -> QuadTwoContexts a %1 -> (a, a, a, a)
-- In the first two cases, we put the 2-hole at the start of the 4-tuple.
fromContext (h1, h2) (Mk (Mk1 y z w) inv) =
    letUnit (inv (Right y)) (h1, h2, z, w)  -- consume y then fill 2-hole

fromContext (h1, h2) (Mk (Mk2 x z w) inv) =
    letUnit (inv (Left x))  (h1, h2, z, w)  -- consume x then fill 2-hole

-- In the second two cases, we put the 2-hole at the end of the 4-tuple.
fromContext (h1, h2) (Mk (Mk3 x y w) inv) =
    letUnit (inv (Right w)) (x, y, h1, h2)  -- consume w then fill 2-hole

fromContext (h1, h2) (Mk (Mk4 x y z) inv) =
    letUnit (inv (Left z))  (x, y, h1, h2)  -- consume z then fill 2-hole
```

The intuition is that the inverse $(2\alpha)^{-1}$ (`inv` above) is used to consume an element of the 4-tuple that overlaps with the hole, with the constructor of `Either` delineating from which position we are consuming.

This technique becomes more useful when we want 2-hole contexts in a type which does not contain an even number of elements–or more generally when we want n-hole contexts from a data type whose number of elements are not exactly divisible by n. For example, we can now compute the type of 5-tuples with 2-holes as:

$$\partial_{(\alpha^2)}\alpha^5 = 5\alpha^4 \otimes (2\alpha \multimap 1)$$

The usual interpretation in the real domain would have yielded $\frac{5}{2}\alpha^3$ for which we have no interpretation in regular types. Instead, we can use the inverses approach to yield contexts of 2-holes for 5-tuples. The resulting equivalent of `fromContext` then has to capture a final hole which overlaps the preceding one, to deal with the fact that 5 is not factored by 2.

An even more interesting possibility presents itself, however. Note that the above example of 2-hole contexts for 4-tuples considers the context to also be chunked into contiguous pairs, and thus we cannot have the context $(x, -_1, -_2, w)$ with the 2-hole "in the middle". However, such an interpretation should certainly be possible using the inverse approach, as there is enough information available: in the domain of $\mathbb{R}$, division (multiplying with an inverse) is a non-injective operation (it destroys information) whereas with regular types, the inverse preserves the structure of the original type until we apply `divide`. Indeed, we can define the following alternate way of mapping the `QuadTwoContexts` data type back to a 4-tuple:

```
fromContext' :: (a, a) %1 -> QuadTwoContexts a %1 -> (a, a, a, a)
fromContext' (h1, h2) (Mk (Mk1 y z w) inv) =
    letUnit (inv (Left h1)) (h2, y, z, w)  -- first hole outside the 4-tuple!

fromContext' (h1, h2) (Mk (Mk2 x z w) inv) =
    letUnit (inv (Left x)) (h1, h2, z, w)  -- 2-hole at start of the 4-tuple

fromContext' (h1, h2) (Mk (Mk3 x y w) inv) =
    letUnit (inv (Left y)) (x, h1, h2, w)  -- 2-hole in middle of the 4-tuple

fromContext' (h1, h2) (Mk (Mk4 x y z) inv) =
    letUnit (inv (Left z)) (x, y, h1, h2)  -- 2-hole at end of the 4-tuple
```

Compared to `fromContext`, this "shifts" the 2-hole through successive positions of the context, not requiring that the surrounding context is broken up into pairs. Instead, `fromContext'` uses the inverse to consume the extra value always under the left component of the hole, leaving the remaining four elements as follows:

$$\cancel{h_1}(h_2, y, z, w) \quad (\cancel{x}\, h_1, h_2, z, w) \quad (x, \cancel{y}\, h_1, h_2, w) \quad (x, y, \cancel{z}\, h_1, h_2)$$

with the consumed element shown here in strikethrough. Note in particular the first case (`Mk1`) where the deleted element is "outside" the 4-tuple; if we have a 2-hole where the second hole is at the leftmost position of a 4-tuple, then the first hole refers to data (h_1) outside the 4-tuple to the left. We can understand such data as being a "boundary value", which hints at an application for n-hole context data types: *stencil computations*.

n-holes and stencil computations

Typically applied to arrays, a stencil computation traverses each position in an array, reading a small neighbourhood of elements at and around the "current" position to compute the corresponding element of a new array. This is used e.g., for image processing (e.g., Gaussian blur), cellular automata, and the finite-difference method for solving PDEs [25]. The idea can be generalised to types other than arrays, e.g., trees, graphs, and triangular meshes [41]. The above notion of a generalised derivative with respect to another type and its interpretation as a kind of zipper on n-holes captures exactly this structure. In the above example of $\alpha^2 \otimes \partial_{\alpha^2}(\alpha^4)$, the 2-hole describes a neighbourhood of two elements, looking in this case "to the left" (in other words, the neighbourhood comprises the current element and one to its left in the above interpretation). In the `Mk1` case above, the value h_1 is the boundary value when the second hole is positioned at the leftmost point. This is standard for stencil computations: we need a "halo" of boundary values to compute at a data structure's edge.

As a more concrete example, consider the discrete Laplace operator over 1-dimensional arrays. Mathematically, we can describe the operation as taking an array A and computing the elements at position i in the output array B as follows:

$$B_i = A_{i-1} - 2A_i + A_{i+1}$$

(Note this ignores what to do at the boundaries of the array at positions A_{-1} and A_{n+1}). This can be structured using a local computation over 3-holes, e.g., in Haskell:

```
1  laplace :: (Float, Float, Float) %1 -> Float
2  laplace (a, b, c) = a - 2*b + c
```

We can then capture the data type of contexts for the corresponding global traversal of a data structure (say lists) by computing the 3-hole contexts, e.g.,

$$\partial_{\alpha^3}(\mathsf{List}\ \alpha) = \partial_\alpha(\mathsf{List}\ \alpha) \otimes (3\alpha^2 \multimap 1)$$

which gives us the usual data type of 1-hole contexts plus a way to consume 2 elements, yielding a gap for 3-holes. A recursive function can then navigate "right" through this data structure, applying `laplace` to every 3-hole to compute the values of an output list, giving one iteration of the discrete Laplace stencil computation. The actual definition of this operation is less relevant to type inverses so we elide it here, but include it in the accompanying code artifact for this pearl. The inverse is then needed to map this zipper data structure back to the original list form, via an operation akin to `fromContext'`.

Thus, our notion of multiplicative inverse has given us a way to generalise derivatives to n-holes, which can then be used to capture the "sliding window" of a stencil computation through any data type.

6 Communicating with Inverses

So far the inhabitants of inverses τ^{-1} have been rather mundane, consuming their inputs by pattern matching. We now turn to a richer setting in which some types have an inverse inhabited by functions which can perform some kind of local side effect. For this we use the functional language Granule which combines linear and indexed types with graded modal types [42], though we will mostly leverage just linear types here. We consider richer inverses first through the question of whether our notion of inverses is an *involution*.

A function is an involution if it is its own inverse, i.e., $f(f(x)) = x$. In abstract algebra, the inverses in groups and fields are automatically involutions.[5] In the setting of Granule, the inverse of a type is also an involution with isomorphism $(\tau^{-1})^{-1} \cong \tau$, which might be surprising given that so far our inverse has been lax.

One direction of the involution isomorphism $\tau \multimap (\tau^{-1})^{-1}$ is easy via function application. We give the definition below in Granule, whose syntax closely resembles Haskell's:

```
1  type Inverse a = a → ()
2
3  -- i.e. the type expands to a → ((a → ()) → ())
4  invol : ∀ {a : Type} . a → Inverse (Inverse a)
5  invol x = λf → f x
```

As one can see, the differences between Granule and Haskell are fairly minimal: Granule's arrows are linear by default (fans of lollipops $\multimap$ will just have to squint in code samples!) whereas in Haskell's linear types extension the linear multiplicity must be explicitly written. The remainder of the translation from Haskell to Granule mainly lies in explicitly quantifying our types and using `:` rather than Haskell's `::` for our type declarations. If the reader would like to follow along using Granule for this section, we recommend the latest release.[6]

Usefully for this pearl, Granule implements a mechanism for algorithmically deriving a weakening operation for regular types [23]; this can be accessed by writing `drop @t` for some type `t`, so we have for example `drop @Bool : Bool → ()`, providing an inverse.

This direction of the involution isomorphism is of course also well-defined in the linear λ-calculus and in Haskell. The opposite direction $(\tau^{-1})^{-1} \multimap \tau$ is much more challenging: in fact it is not inhabited if we restrict ourselves to the linear λ-calculus or even traditional non-linear Haskell, but it is instead a *sequentially-realizable function* [32, 33].

A sequentially-realizable function is one which has outwardly pure behaviour but relies on a notion of local side effects; these are entirely contained within the body of the function. Traditionally, sequentially-realizable functions have only been expressible in the ML-family of languages (which allow unrestricted side effects) but not at all in Haskell. However, in the context of linear typing, we can now safely re-introduce some side effects via linear references or linear channels. We show the latter approach, leveraging Granule's session-typed linear channels [35] inspired by the GV calculus [54]. The same approach would work equally well in other implementations of similar calculi, such as FST [31] or FREEST 2 [3].

Originating from Gay and Vasconcelos [17], further developed by Wadler [54], for which we use the formulation of Lindley and Morris [30], the GV calculus extends the linear λ-calculus with a type of channels parameterised by session types [57], which capture the protocol of interaction allowed over the channel. Granule provides an analogous type of linear channels `LChan : Protocol → Type` indexed by protocols, given by the constructors:

---

[5] For a group $(X, \bullet, e)$ then the properties of inverses yield $(x^{-1})^{-1} \bullet x^{-1} = e$ which implies $(x^{-1})^{-1} \bullet x^{-1} \bullet x = x$; thus, by the same properties we have $(x^{-1})^{-1} = x$.

[6] Examples were tested on `https://github.com/granule-project/granule/releases/tag/v0.8.1.0`

```
Send : Type → Protocol → Protocol             End : Protocol
Recv : Type → Protocol → Protocol
```

The following primitives are then provided for using these (synchronous) channels:

```
send       : ∀ {a : Type, s : Protocol} . LChan (Send a s) → a → LChan s
recv       : ∀ {a : Type, s : Protocol} . LChan (Recv a s) → (a, LChan s)
close      : LChan End → ()
forkLinear : ∀ {s : Protocol} . (LChan s → ()) → LChan (Dual s)
```

This is a subset of the available primitives. We can see that `send` takes an input channel with protocol `Send a s` and an input value `a` which is sent over the channel to yield a channel which can be used according to protocol `s`. The `recv` function is dual, taking a channel which is allowed to follow protocol `Recv a s`, returning a pair of the received `a` value and a new channel that can behave as `s`. The `close` primitive consumes a channel which is at the end of a protocol. Lastly, `forkLinear` spawns a process from the parameter function, which is applied to a freshly created channel, returning a channel with the "dual" protocol in order to communicate with the new process. Here, `Dual` is a type-level function defined:

```
Dual (Send a s) = Recv a (Dual s)             Dual End = End
Dual (Recv a s) = Send a (Dual s)
```

Interestingly, `forkLinear` is a combinator relating inverse and duality: $(\texttt{LChan s})^{-1} \multimap \texttt{LChan}\ (\texttt{s}^{\perp})$, that is, a function consuming a channel with behaviour `s` yields a channel with dual behaviour (denoted by the standard notation $\perp$ as in linear logic and the GV calculus).

We now have adequate machinery to define involution in the direction $(\tau^{-1})^{-1} \multimap \tau$:

```
1 involOp : ∀ {a : Type} . Inverse (Inverse a) → a
2 involOp k =
3   let r       = forkLinear (λs → k (λx → let c = send s x in close c));
4       (x, c') = recv r;
5       ()      = close c'
6   in x
```

Thus `k` has type `(a → ()) → ()`, to which the function `λx → let c = send s x in close c` is passed; this sends the input `x : a` on the channel `c` which is then closed. This channel is provided by `forkLinear`, and so `k` is applied in a process taking one end of the channel to "sneak out" the value of type `a`. Outside this, `recv` waits to receive from the dual end of the channel returned by `forkLinear`. The remaining channel is closed and `x` is returned. A local side effect is performed within `involOp` which is not observable externally, but it would not have been possible to construct the required function without carrying out this effect.

In languages such as ML, an alternate equivalent definition can be given using mutable references which also resembles Longley's **F** combinator [32].

Lastly, the functions `invol` and `involOp` form an isomorphism, witnessing $\tau \cong (\tau^{-1})^{-1}$. The proof of this can be shown via calculating on the definitions, with details given in Appendix B. We refer the reader to Lindley et al. [30, 31] for details of how adding session-typed linear channels to the linear λ-calculus (and linear System F) retains type safety.

Session types based on linear channels can also be represented in Haskell, but they do not allow us to demonstrate a full isomorphism to the same extent. Here we illustrate the more challenging direction of the involution using the Priority Sesh library,[7] a recent package for

[7] Available at `https://github.com/wenkokke/priority-sesh`

session-typed communication in Linear Haskell which is itself inspired by the GV calculus [27]. However, note that the two directions cannot quite form an involution here, since everything must be wrapped up in the linear IO monad for these session types to be used; we cannot confine the side effects to the function as is possible in Granule.

```
type Inverse' a = a %1 -> Linear.IO ()

involOp :: Inverse' (Inverse' a) %1 -> Linear.IO a
involOp k = do
  (s, r) <- new
  void $ forkIO $ k (\z -> send s z)
  recv r
```

Continuation monad

A double inverse type $(\tau^{-1})^{-1} = (\tau \multimap 1) \multimap 1$ is also a specialisation of the familiar *continuation monad* [52], whose return type is the unit type (the Haskell data type is often written `data Cont r a = Cont ((a -> r) -> r)` so this is `Cont () a` here). The definition of `invol` provides the `return` operation of the monad and the "bind" operator is the usual definition for the continuation monad:

```
return : ∀ {a : Type} . a → Inverse (Inverse a)
return = invol

bind : ∀ {a b : Type}
     . Inverse (Inverse a) → (a → Inverse (Inverse b)) → Inverse (Inverse b)
bind m k = λc → m (λa → k a c)
```

A standard way of understanding the use of the continuation monad is to see that its Kleisli arrows (functions of type `a → Inverse (Inverse b)`) characterise continuation-passing style (CPS) programs which can then be sequentially composed. This can be seen via a little algebraic manipulation:

```
a → Inverse (Inverse b)  ≡  a → ((b → ()) → ())  ≅  (b → ()) → (a → ())
```

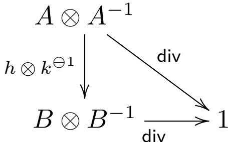
Thus we can see that the Kleisli arrows are CPS-transformed functions of "$a \to b$". Under our interpretation, these are the same as functions that map consumers of b to consumers of a, and the "double inverse" monad gives us a sequential composition for these inverses.

The usual way to "evaluate" a continuation monad computation is to end up with a value of type `Cont r r` (i.e., `(r -> r) -> r`) to which the identity is applied to return the "final" value of type `r`. This requires that the `r` type of the whole `Cont r` computation is pre-determined based on what value we want to be able to extract from a continuation monad computation. For `Inverse (Inverse a)` we can only apply the identity when `a` = `()`, i.e., only in trivial cases. However, our "double inverse" monad is actually more powerful thanks to linearity and sequential realizability: we can extract the value of any `Inverse (Inverse a)` computation for any `a` by applying the sequentially-realizable involution function `involOp` to extract the `a` value. This ends up being more flexible than the continuation monad since we need not pre-determine the continuation "result" type (which for the double inverse is fixed as unit anyway) and we can extract the value inside the computation at any point.

Thus, viewing the continuation monad through the lens of linear-logical inverses yields a more flexible continuation-passing style monadic composition. The crucial restriction, though, is that the continuations must be used *linearly*.

Calculating with inverses that communicate

Recall the naturality property discussed in Section 2:



In Granule with local side effects due to channels, this equation only holds if $k \circ h = id$. Consider the following code where `divNat` captures the left-bottom path of the diagram:

```
1 divNat : ∀ {a b : Type} . (a → b) → (b → a) → (a, Inverse a) → ()
2 divNat h k (x, y) = divide (h x) (comap k y)
3
4 example : ∀ {a b : Type} . (a → b) → (b → a) → a → a
5 example h k a =
6   let r = forkLinear (λs → divNat h k (a, λy → let c = send s y in close c));
7       (a', c') = recv r;
8       () = close c'
9   in a'
```

The `example` function applies `divNat` inside a forked process where the inverse sends the result on the channel which is received on the outside. `example h k` is only the identity function if `k . h` $= id$, e.g., `example (λx→ x + 1) (λx→ x - 1) 42` evaluates to `42`. Thus, we can see the power of the local side effects; here inverses can do more than just consume.

7 Additive Inverses

As we have demonstrated, a reasonable definition of inverses exists for product types in the realm of linear logic. One might therefore wonder whether defining inverses for sum types (i.e., *subtraction*) is also feasible. In much the way that defining a multiplicative inverse almost gives us a semifield of types because some of the identities are lax, being able to define an additive inverse would similarly give us something closely approximating a ring of types. The answer as to whether we can do this, however, is somewhat mixed.

The linear regular types we have been using have product types as linear logic's "multiplicative conjunction" $\otimes$ and sum types as "additive disjunction" $\oplus$. Unfortunately, we cannot define a sensible additive inverse for this operator. To do so we would need to have $A \oplus -A \cong 0$. However, defining a map $A \oplus -A \multimap 0$ is impossible regardless of the value of $-A$, because if A is nonempty then $A \oplus B$ must also be nonempty which means we cannot construct a term of type 0. Furthermore, defining a map $0 \multimap A \oplus -A$ is impossible unless $A = -A = 0$, as otherwise we would have to be able to construct some value of either type A or type $-A$ from nothing. Consequently, linear regular types cannot form a *field* (with both multiplicative and additive inverses).

It turns out that the reason it is not possible to define an additive inverse in the context of standard intuitionistic logic or while using the $\oplus$ operation offered by linear regular types is a corollary to a result known as *Crolard's lemma* [5]. This lemma states that subtraction $A \setminus B$ cannot be defined for disjoint unions in the category of sets unless either A or B is the empty set. In fact, this result also applies to any operation like $\oplus$ which allows a free choice between A and B, so any definition of subtraction with respect to $\oplus$ must be trivial.

However, defining products as multiplicative conjunction and sums as additive disjunction as in regular types is not the only possible interpretation we can use for these concepts. We can just as easily have product types that follow the rules of the & operator, pronounced “with” and describing “additive conjunction”, and similarly we can have sum types based on the $\parr$ operator, pronounced “par”, which describes “multiplicative disjunction”. This makes it possible to discuss a closely related setting which we will call *coregular types* (as these operations behave in a dual manner to those used to define *regular types*), with type syntax:

$$\tau ::= \tau \,\&\, \tau' \mid \tau \parr \tau' \mid \top \mid \bot$$

where $\top$ and $\bot$ are the units for & and $\parr$ respectively. The intuition for & is that $a \,\&\, b$ allows us to select one of a or b and use it, rather than having access to both at the same time but having to use each one, as with $a \otimes b$. The $\parr$ operator is more difficult to understand intuitively, but one interpretation is that $a \parr b$ gives two processes a and b that happen in parallel, and we have the choice of how to interleave the two processes [4]. The important thing to keep in mind is that & is dual to $\oplus$ and $\parr$ is dual to $\otimes$.[8]

If we consider the coregular $\parr$ sum types which do not allow a free choice between A and B rather than the regular $\oplus$ sum types, then the outlook for defining an additive inverse is less bleak: it is possible to define an inverse operation to multiplicative disjunction. Cointuitionistic linear logic [5] offers an operation called *linear subtraction*, denoted $A \setminus B$ and read “A *excludes* B”, which acts as the left adjoint of $\parr$. Intuitively, we can understand linear implication in the following way:

$$A \otimes B \vdash C \text{ if and only if } A \vdash B \multimap C$$

which arises from the categorical notion of adjunctions: $(- \otimes B)$ is left adjoint to $(B \multimap -)$. Dually, linear subtraction can be understood as follows:

$$A \vdash B \parr C \text{ if and only if } A \setminus B \vdash C$$

In other words, if A gives us $B \parr C$ then A *excluding* the possibility of B gives us C, and conversely if A excluding B gives C then from A we can get $B \parr C$.

In this dual setting, we must now find a definition of subtraction from a suitable unit which acts as the additive inverse we desire. Since linear subtraction is dual to linear implication, just as we can define implication in terms of the $\parr$ connective (i.e. $A \multimap B \equiv A^\perp \parr B$), we can similarly represent subtraction using the $\otimes$ connective, as $B \setminus A \equiv B \otimes A^\perp$.

Using this representation of linear subtraction, by duality we can show that a nontrivial inverse to multiplicative disjunction $\parr$ exists in the context of linear type theory. Dually to our definition of an inverse for multiplicative conjunction, we can define an inverse to multiplicative disjunction as $-\tau \triangleq \bot \setminus \tau$, via linear subtraction as discussed above.

Similarly to the lax identity $\tau^{-1} \otimes \tau \multimap 1$, the additive inverse $-\tau$ also satisfies a lax inverse law, but in the opposite direction:

$$\bot \multimap \tau \parr -\tau \tag{8}$$

Via the identity between $\multimap$ and $\parr$ and between $\setminus$ and $\otimes$ then $\tau \parr -\tau \cong \tau^\perp \multimap (\bot \otimes \tau^\perp)$. If we had a type system involving both regular and coregular types with a duality operator the above lax law (8) would be given constructively by the term $\lambda b.(\lambda x.(b, x)) : \bot \multimap \tau^\perp \multimap (\bot \otimes \tau^\perp)$.

[8] Classical linear logic has an involutive duality operator, written $(-)^\perp$, where $(A \,\&\, B)^\perp = A^\perp \oplus B^\perp$ and $(A \parr B)^\perp = A^\perp \otimes B^\perp$.

The applications of this identity are less apparent, as we cannot construct a witness for it in the same way as the lax inverse law from Section 2 given the constraints of having to choose between working with either regular or coregular types. If we were not constrained by this limitation, we could have an algebraic structure with all four common mathematical operations, with $\bot$ acting as an additive identity and 1 acting as a multiplicative identity.[9] Intuitionistic and cointuitionistic logic can be combined into a single framework, known as bi-intuitionistic logic, and work on making sense of this through the lenses of type theory and category theory is ongoing [15]; this could provide a way to combine regular and coregular types in a single system.

Given the above definitions we can show a lax involution in one direction for additive inverses. Between multiplicative conjunction and disjunction there is a distribution: $(A \otimes (B \parr C)) \multimap ((A \otimes B) \parr C)$ which is not an isomorphism, but it is a valid implication in this direction [13]. Using this weak distribution, for all τ this lax involution is defined:

$$
\begin{aligned}
-(-\tau) &\cong (\bot \setminus (\bot \setminus \tau)) \\
&\cong \bot \otimes (\bot \otimes \tau^{\bot})^{\bot} \\
&\cong \bot \otimes (1 \parr \tau) \\
&\multimap (\bot \otimes 1) \parr \tau \\
&\cong \bot \parr \tau \\
&\cong \tau
\end{aligned}
$$

Interestingly, this kind of dichotomy between additive and multiplicative disjunction can also be seen for conjunction. The multiplicative inverse we have defined for linear regular types does not behave well if we attempt to apply it to the & operator (additive conjunction). We *cannot* define a map $A \,\&\, (A \multimap 1) \multimap 1$, because we can only use one of the two components of the & on the left so we cannot apply the inverse to the A value. Furthermore, similarly to $\otimes$, we cannot in general define a map in the other direction as we would need to be able to construct a value of an arbitrary type A from nothing.

In the end, linear logic cannot yet give us a field of types – it can only afford to give us a ring or half a field (a semifield), but both at once is beyond our current budget. The semifield interpretation however has the closest intuitions to familiar concepts in functional programming, and linear regular types are certainly more frequently encountered than coregular ones in the current programming landscape, hence our focus on them in this pearl.

8 Discussion: Thinking with Inverses

As we near the end of our journey, we remark on some alternate perspectives and approaches, and some connections with related work.

Curry-Howard with inverses

From a logical standpoint, the (lax) inverse property we have discussed provides a natural notion of inverse elimination and introduction in a natural deduction logic for a Curry-Howard correspondent to a type's inverse:

---

[9] We still cannot, however, form a field even if we use $\otimes$ and $\parr$ as our operations; they do not obey distributivity, $A \otimes \bot \ncong \bot$ (note for example that $\top \otimes \bot \cong \top$), and indeed both types of inverse only obey lax inverse laws, so the required isomorphisms for a field do not exist.

$$\frac{\Gamma \vdash p \qquad \Gamma \vdash p^{-1}}{\Gamma, \Gamma' \vdash 1}\ {}^{-1}E \qquad \frac{\Gamma, p \vdash 1}{\Gamma \vdash p^{-1}}\ {}^{-1}I$$

i.e., elimination is just a specialised modus ponens and an inverse p^{-1} is introduced by a (linear) proof starting with p and concluding 1 – the subproof consumes the assumption p.

Duality

One may consider trying to use the classical linear logic notion of "duality" $\tau^{\perp}$ [20] to provide multiplicative inverses, but it does not behave accordingly: $1 \otimes 1^{\perp} = 1 \otimes \perp = \perp$ but instead we would like $1 \otimes 1^{-1} \cong 1$. However, there are various interesting applications of linear and classical duality relating call-by-value and call-by-name [53, 45]. These take advantage of various properties of duality, some of which our inverses do indeed share.

Negative and fractional types

Despite the algebraic manipulation of data types producing a rich source of ideas, inverses appear to have not had much consideration. One notable thread though is due to James and Sabry, who consider *negative* and *fractional* types in the context of reversible computations where a reciprocal $1/b$ "*imposes constraints on [its] context*" acting as a logical variable [24]. They present a reversible calculus admitting isomorphisms $\eta : 1 \leftrightarrow (1/b) \times b$ for all types: with left-to-right direction producing a fresh logical variable α inhabiting b along with its dual, and the inverse η^{-1} corresponding to unification of logical variables. Later they interpret this categorical semantics computationally [12], defining a sound operational semantics for such types, in which a negative type represents a computational effect that "reverses execution flow" and a fractional type represents one that "garbage collects" values or throws exceptions. This differs from our approach but certainly has some of the same flavour. We cannot however construct a pair of a $b^{-1} \otimes b$ out of thin air for any b.

Cardinalities

As recalled in the introduction, the cardinality operation on types is a semiring homomorphism from regular types to natural numbers (e.g., $|a \times b| = |a||b|$). So what is the cardinality of an inverse type? In a Cartesian setting a function $\tau \to 1$ simply has cardinality 1 since $|\tau \to 1| = |1|^{|\tau|} = 1$. In a linear setting without side effects (i.e., linear channels), we can recover a similar result. As a simple example, consider the boolean type. The cardinality of `Bool` is 2 as it has two elements: `True` and `False`. The cardinality of `Inverse Bool`, on the other hand, is 1; the type has exactly one inhabitant: the `boolDrop` function shown in Section 2.

We could however consider a different notion of cardinality that would allow for $|\tau^{a}| \times |\tau^{b}| = |\tau^{a+b}|$ in general; this statement already holds for $a \geq 0 \wedge b \geq 0$, but now we consider cases where the types are not necessarily isomorphic. In particular, we can examine the notion of *fractional cardinalities* [44, 46] assigning a generalised cardinality of $(1/|\tau|)^{m}$ to τ^{-m}.

If we specialise this we can let $a = 1$ and $b = -1$ and see that $|\tau \otimes \tau^{-1}| = |\tau| \times |\tau^{-1}| = n \times (1/n) = 1 = |1|$; the two types have the same fractional cardinality even though we only have a lax map from $\tau \otimes \tau^{-1}$ to 1. Of course, this does not match up with the standard idea of cardinality on types, as it is clear that $\tau \otimes \tau^{-1}$ has at least as many inhabitants as τ. We leave an interpretation for this as something for others to ponder.

Taylor series

As we have seen, it is not possible to form a field out of regular types (linear or otherwise), because the additive operation that permits an inverse is not the same addition which behaves like the logical $\oplus$ we would usually want. But it turns out that if we suspend our disbelief and assume that types do form a field, some results from real analysis can be applied with surprising success: Taylor series approximations can yield solutions to recursive types. Consider the recursive definition of lists over elements of type α:

$$\mathsf{List}\ \alpha = 1 \oplus (\alpha \otimes \mathsf{List}\ \alpha)$$

Through some unjustified algebraic rearrangement we get $\mathsf{List}\ \alpha = \frac{1}{(1-\alpha)}$ on which we can compute the Taylor expansion yielding the familiar least fixed-point solution of $\mathsf{List}\ \alpha$:

$$\mathsf{List}\ \alpha = 1 \oplus \alpha \oplus \alpha^2 \oplus \alpha^3 \oplus \ldots$$

i.e. a list is either empty, or has one element, or has two elements, etc.

This is quite surprising; we must apply the equations of a field to yield a derivation for this result, using inverses we do not have access to in the realm of regular types, and yet we end up with a result that makes sense using only regular operations. Whether there is an interpretation of our inverses that can lend a meaningful foundation to these intermediate manipulations is unclear, but would certainly be interesting to look into.

One might wonder whether it is coincidental that this result happens to hold true for lists in particular, but this is not the case. Fiore and Leinster [16] show that, for all complex numbers t and polynomials p, q_1 and q_2 with non-negative coefficients (with some restrictions), then if $t = p(t)$ implies $q_1(t) = q_2(t)$ the same result also holds up to isomorphism in any other semiring (as well as for complex numbers), which includes regular types.

This was applied to great effect for the example of finite binary trees to demonstrate the famous "seven trees in one" result [9], showing that there is a particularly elementary bijection (involving case distinctions only down to a fixed depth) between the set T of finite binary trees and the set T^7 of 7-tuples of such trees. It is more difficult to find solutions to more complex types via this kind of equational reasoning, though, particularly due to the Abel-Ruffini theorem [2] which states that there is no solution in radicals to general polynomial equations of degree five or higher.

9 Epilogue

Summary

Taking $\tau^{-1} \triangleq \tau \multimap 1$ yields a useful notion of multiplicative inverse for linear regular types. We have seen this yields (lax) exponentiation laws in the presence of negative coefficients:

$$\tau \otimes \tau^{-1} \multimap 1 \qquad \tau^{a} \otimes \tau^{-b} \multimap \tau^{a-b} \qquad \tau^{-a} \otimes \tau^{-b} \cong \tau^{-(a+b)}$$
$$\sigma^{-a} \otimes \tau^{-a} \multimap (\sigma \otimes \tau)^{-a} \qquad \tau \multimap (\tau^{-1})^{-1}$$

for all $a \geq 0, b \geq 0$. The first lax identity is generalised by the second (Section 4). The fourth is induced by $-^{-1}$ being a monoidal functor (Section 3). The last lax identity becomes an isomorphism $\tau \cong (\tau^{-1})^{-1}$ when sequentially realizable functions are permitted (Section 6).

Fin

The algebraic characteristics of data types have been studied and leveraged since the dawn of functional programming; we call them "algebraic" data types, after all. In the linear setting, the idea of consumption as a lax multiplicative inverse has given us a fresh perspective on the algebraic characterisation of regular linear types. Now that linear typing is becoming more mainstream, e.g., in Haskell [6], and with closely related ideas arising in languages like Clean and Rust (the concept of uniqueness which is in some sense dual to linearity [21, 14, 36], and more sophisticated systems tracking ownership and borrowing [56]), this is now an ideal time to start taking our algebraic understanding of data types to the next level. This pearl has been a demonstration of how one weird trick can lead to a journey through many interesting and diverse areas of our field. We hope that that this has stoked your enthusiasm for investigating the idea of taking the inverse of a type even further.

References

1. Michael Abbott, Thorsten Altenkirch, Conor Mcbride, and Neil Ghani. ∂ for data: Differentiating data structures. *Fundam. Inf.*, 65(1–2):1–28, January 2005.
2. Niels Henrik Abel. Mémoire sur les equations algébriques, où l'on démontre l'impossibilité de la résolution de l'équation générale du cinquième degré. 1:28–33, 1824. `doi:10.1017/CBO9781139245807.004`.
3. Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic context-free session types, 2021. `arXiv:2106.06658`.
4. Federico Aschieri and Francesco A. Genco. Par means parallel: Multiplicative linear logic proofs as concurrent functional programs. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. `doi:10.1145/3371086`.
5. Gianluigi Bellin, Massimiliano Carrara, Daniele Chiffi, and Alessandro Menti. Pragmatic and dialogic interpretations of bi-intuitionism. Part I. *Logic and Logical Philosophy*, 23(4):449–480, 2014.
6. Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.
7. Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall, Inc., USA, 1997.
8. Richard S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.
9. Andreas Blass. Seven trees in one. *Journal of Pure and Applied Algebra*, 103(1):1–21, 1995.
10. Edwin Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2021.9`.
11. Jacques Carette and Amr Sabry. Computing with semirings and weak rig groupoids. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 123–148, Berlin, Heidelberg, 2016. Springer-Verlag.
12. Chao-Hong Chen and Amr Sabry. A computational interpretation of compact closed categories: Reversible programming with negative and fractional types. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. `doi:10.1145/3434290`.
13. J.R.B. Cockett and R.A.G. Seely. Weakly distributive categories. *Journal of Pure and Applied Algebra*, 114(2):133–173, 1997. `doi:10.1016/0022-4049(95)00160-3`.
14. Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness typing simplified. In *Symposium on Implementation and Application of Functional Languages*, pages 201–218. Springer, 2007. `doi:10.1007/978-3-540-85373-2_12`.

15 Harley Eades III and Gianluigi Bellin. A cointuitionistic adjoint logic, 2017. arXiv:1708.05896.
16 Marcelo Fiore and Tom Leinster. Objects of categories as complex numbers. *Advances in Mathematics*, 190(2):264–277, 2005. doi:10.1016/j.aim.2004.01.002.
17 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.
18 Jeremy Gibbons. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 151–203. Springer, 2002.
19 Jeremy Gibbons. The school of Squiggol - A history of the Bird-Meertens formalism. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*, pages 35–53, 2019. doi:10.1007/978-3-030-54997-8_2.
20 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
21 Dana Harrington. Uniqueness logic. *Theoretical Computer Science*, 354(1):24–41, 2006.
22 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
23 Jack Hughes, Michael Vollmer, and Dominic Orchard. Deriving distributive laws for graded linear types. In *TLLA/Linearity*, 2020.
24 Roshan P James and Amr Sabry. The Two Dualities of Computation: Negative and Fractional Types. Technical report, Indiana University, 2012.
25 Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 51–60, 2006.
26 G Maxwell Kelly. Coherence theorems for lax algebras and for distributive laws. In *Category seminar*, pages 281–375. Springer, 1974.
27 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, pages 1–13, 2021.
28 Serge Lang. *Algebra*. Springer, New York, NY, 2002.
29 Gottfried Wilhelm Leibniz. Nova methodus pro maximis et minimis, itemque tangentibus, qua nec irrationals quantitates moratur. *Acta eruditorum*, 1684.
30 Sam Lindley and J Garrett Morris. A semantics for propositions as sessions. In *European Symposium on Programming Languages and Systems*, pages 560–584. Springer, 2015.
31 Sam Lindley and J Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools. River Publishers*, pages 265–286, 2017.
32 John Longley. When is a functional program not a functional program? In *ACM SIGPLAN Notices*, volume 34(9), pages 1–7. ACM, 1999.
33 John Longley. The sequentially realizable functionals. *Ann. Pure Appl. Log.*, 117(1-3):1–93, 2002. doi:10.1016/S0168-0072(01)00110-5.
34 Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5. Springer Science & Business Media, 2013.
35 Daniel Marshall and Dominic Orchard. Replicate, reuse, repeat: Capturing non-linear communication via session types and graded modal types. *Proceedings of PLACES 2022, Electronic Proceedings in Theoretical Computer Science*, 356:1–11, March 2022. doi:10.4204/eptcs.356.1.
36 Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and Uniqueness: An Entente Cordiale. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 346–375, Cham, 2022. Springer International Publishing.
37 Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.
38 Conor McBride. Clowns to the left of me, jokers to the right (pearl): Dissecting data structures. *SIGPLAN Not.*, 43(1):287–295, January 2008. doi:10.1145/1328897.1328474.
39 J. Garrett Morris. The best of both worlds: linear functional programming without compromise. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 448–461. ACM, 2016. doi:10.1145/2951913.2951925.

40 Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *International Workshop on Types for Proofs and Programs*, pages 252–267. Springer, 2004.
41 Dominic Orchard. Programming contextual computations. Technical report, University of Cambridge, Computer Laboratory, 2014.
42 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, 2019.
43 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, Gothenburg, Sweden, September 1-3, 2014*, pages 123–135, 2014. doi:10.1145/2628136.2628160.
44 James Propp. Euler measure as generalized cardinality. *arXiv: Combinatorics*, 2002.
45 Ben Rudiak-Gould, Alan Mycroft, and Simon Peyton Jones. Haskell is not not ML. In *European Symposium on Programming*, pages 38–53. Springer, 2006.
46 Stephen H. Schanuel. Negative sets have Euler characteristic and dimension. In Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini, editors, *Category Theory*, pages 379–385, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
47 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Science of Computer Programming*, 78(8):1176–1192, 2013. doi:10.1016/j.scico.2012.09.005.
48 Kornel Szlachányi. Skew-monoidal categories and bialgebroids. *Advances in Mathematics*, 231(3-4):1694–1730, 2012.
49 Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458, 2011. doi:10.1145/1926385.1926436.
50 Tarmo Uustalu, Niccolò Veltri, and Noam Zeilberger. The sequent calculus of skew monoidal categories. *Electronic Notes in Theoretical Computer Science*, 341:345–370, 2018.
51 Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, volume 3(4), page 5. Citeseer, 1990.
52 Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1992.
53 Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 189–201, 2003.
54 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.
55 David Walker. Substructural type systems. *Advanced Topics in Types and Programming Languages*, pages 3–44, 2005.
56 Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of Rust, 2021. arXiv:1903.00982.
57 Nobuko Yoshida and Vasco T Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, 2007.

A Regular Linear Types

The type syntax for linear regular types (as discussed in Section 1) is as follows.

$$\tau ::= \tau \otimes \tau' \mid \tau \oplus \tau' \mid 1 \mid 0 \mid X \mid \mu X.\tau$$

where X ranges over recursion variables. We mostly focus on the non-recursive subset (just the first four constructs above), although recursive types make an appearance in Section 5 and we include them here for coherence with the usual description of regular types in the literature. Throughout we use τ and σ to range over types and also A, B, C, D.

The typing rules for linear regular types are as follows, which includes their standard term formers.

$$\frac{}{x : A \vdash x : A}\text{VAR} \quad \frac{\Gamma \vdash t : A \quad \Delta \vdash t' : B}{\Gamma, \Delta \vdash (t, t') : A \otimes B}\otimes\text{I} \quad \frac{\Gamma \vdash t : A \otimes B \quad \Delta, u : A, v : B \vdash t' : C}{\Gamma, \Delta \vdash \mathbf{let}\ (u, v) = t\ \mathbf{in}\ t' : C}\otimes\text{E}$$

$$\frac{}{\vdash * : 1}1\text{I} \quad \frac{\Gamma \vdash t : 1 \quad \Delta \vdash t' : C}{\Gamma, \Delta \vdash \mathbf{let}\ () = t\ \mathbf{in}\ t' : C}1\text{E} \quad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{inl}\ t : A \oplus B}\oplus\text{I}_L \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{inr}\ t : A \oplus B}\oplus\text{I}_R$$

$$\frac{\Gamma \vdash t : A \oplus B \quad \Delta, u : A \vdash t' : C \quad \Delta, v : B \vdash t'' : C}{\Gamma, \Delta \vdash \mathbf{case}\ t\ \mathsf{of}\ \mathsf{inl}\ u \to t' \mid \mathsf{inr}\ v \to t'' : C}\oplus\text{E}$$

Note that in the above we do not include the linear function space $\tau \multimap \tau'$ since we considered just the syntax of regular types in Section 1, but linear functions are used throughout the paper. Their introduction and elimination rules are:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B} \quad \frac{\Gamma \vdash t : A \multimap B \qquad \Delta \vdash t' : A}{\Gamma, \Delta \vdash t\ t' : B}$$

As stated in Section 1, (linear) regular types behave *like* a commutative semiring, i.e., $\otimes$ and $\oplus$ are both commutative and associative with 1 and 0 as their corresponding units, with distributivity, but all up to isomorphism.

$$\begin{array}{ccc} A \otimes (B \otimes C) \cong (A \otimes B) \otimes C & A \oplus (B \oplus C) \cong (A \oplus B) \oplus C & A \otimes (B \oplus C) \cong (A \otimes B) \oplus (A \otimes C) \\ A \otimes B \cong B \otimes A & A \oplus B \cong B \oplus A & (B \oplus C) \otimes A \cong (B \otimes A) \oplus (C \otimes A) \\ 1 \otimes A \cong A & A \oplus 0 \cong A & A \otimes 0 \cong 0 \end{array}$$

All of the above isomorphisms are witnessed by pairs of mutually inverse functions.

Regular types also permit a notion of (positive) exponent, with τ^a defined inductively as:

$$\tau^0 = 1 \qquad \tau^{a+1} = \tau \otimes \tau^a$$

The usual positive exponent laws then hold up to isomorphism via associativity and commutativity (and removal of units in the case of the leftmost isomorphism), for all $a \geq 0, b \geq 0$:

$$\tau^1 \cong \tau \qquad \tau^a \otimes \tau^b \cong \tau^{a+b} \qquad (\tau^a)^b \cong \tau^{ab} \qquad (\sigma \otimes \tau)^a \cong \sigma^a \otimes \tau^a$$

A.1 Equations

This calculus has equations for (bi)functoriality of $\otimes$ and $\oplus$:

$$\begin{array}{rlrl} id \otimes id &= id & id \oplus id &= id \\ (f \otimes g) \circ (h \otimes k) &= (f \circ h) \otimes (g \circ k) & (f \oplus g) \circ (h \oplus k) &= (f \circ h) \oplus (g \circ k) \end{array}$$

The following equations are on the interaction of cotupling, injections and the $\oplus$ bifunctor, which are subset of those from Gibbons [18] that are derivable for the coproduct part of linear regular types:

$$\begin{array}{rlrlrl} [f, g] \circ \mathsf{inl} &= f & [h \circ \mathsf{inl}, h \circ \mathsf{inr}] &= h & [f, g] \circ (h \oplus k) &= [f \circ h, g \circ k] \\ [f, g] \circ \mathsf{inr} &= g & [\mathsf{inl}, \mathsf{inr}] &= id & h \circ [f, g] &= [h \circ f, h \circ g] \end{array}$$

The usual axioms for a (lax) monoidal functor hold for the (contravariant) inverse functor. These axioms are as follows:

$$\begin{array}{ll} \mathtt{mmult} \circ (\mathtt{munit} \otimes id) \circ \lambda_i = \lambda^{\ominus 1} & : A^{-1} \multimap (1 \otimes A)^{-1} \\ \mathtt{mmult} \circ (id \otimes \mathtt{munit}) \circ \rho_i = \rho^{\ominus 1} & : A^{-1} \multimap (A \otimes 1)^{-1} \\ \mathtt{mmult} \circ (\mathtt{mmult} \otimes id) \circ \alpha_i = \alpha^{\ominus 1} \circ \mathtt{mmult} \circ (id \otimes \mathtt{mmult}) & : A^{-1} \otimes (B^{-1} \otimes C^{-1}) \multimap ((A \otimes B) \otimes C)^{-1} \end{array}$$

where α is associativity and $\lambda : 1 \otimes A \multimap A$ and $\rho : A \otimes 1 \multimap A$ (and their inverses λ_i and ρ_i) witness the unit properties of $\otimes$.

B Involution is an Isomorphism

We show that for all τ, then τ^{-1} is a sequentially realizable involution up to isomorphism, i.e., $(\tau^{-1})^{-1} \cong \tau$, with $\tau \multimap ((\tau \multimap 1) \multimap 1)$ implemented as $\lambda x.\lambda f.f\ x$ (see Section 6) and the converse direction as follows, using the syntax of GV calculus (as formulated by [30]) rather than Granule as shown in Section 6, of type $((\tau \multimap 1) \multimap 1) \multimap \tau$:

$$(\lambda k.\textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.k(\lambda x.\mathsf{send}\ c\ x)))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x)$$

In order to prove that τ^{-1} is an involution up to isomorphism, we need to show that the functions $i : \tau \multimap ((\tau \multimap 1) \multimap 1$ and $j : ((\tau \multimap 1) \multimap 1) \multimap \tau$ are mutually inverse, or in other words that $j(i(t)) = t : \tau$ and $i(j(h)) = h : (\tau \multimap 1) \multimap 1$.

We leverage the $\beta\eta$-equality theory of GV based on its operational semantics given by [30], which is the same operational semantics for channels implemented in Granule [42].

We show both directions separately, as follows:

$$\begin{aligned}
j(i(t)) &= (\lambda k.\textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.k(\lambda x.\mathsf{send}\ c\ x)))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x)(\lambda f.f\ t)\\
&= (\textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.(\lambda f.f\ t)(\lambda x.\mathsf{send}\ c\ x)))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x)\\
&= (\textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.(\lambda x.\mathsf{send}\ c\ x)t))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x)\\
&= (\textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x)
\end{aligned}$$

Checking the typing of the inner expression, we have:

$$\begin{aligned}
\lambda c.\mathsf{send}\ c\ t &: \mathsf{Chan}(!\tau.\mathsf{end}_!) \multimap \mathsf{Chan}(\mathsf{end}_!)\\
\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t) &: \mathsf{Chan}(?\tau.\mathsf{end}_?)\\
\mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t)) &: \tau \otimes \mathsf{Chan}(\mathsf{end}_?)
\end{aligned}$$

So we have the binding $(x,c) : \tau \otimes \mathsf{Chan}(\mathsf{end}_?)$. Applying the global configuration semantics of GV [30, Figure 4], we then get the following:

$$\begin{aligned}
& \textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x)\\
(\textsc{Lift+Fork}) \rightsquigarrow\ & (\nu \mathsf{c})(\textbf{let }(x,c) = \mathsf{recv}\ \mathsf{c}\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x) \mid (\mathsf{send}\ \mathsf{c}\ t)\\
(\textsc{Lift+Send}) \rightsquigarrow\ & (\nu \mathsf{c})(\textbf{let }(x,c) = (t,\mathsf{c})\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x) \mid \mathsf{c}\\
(\textsc{LiftV}) \rightsquigarrow\ & (\nu \mathsf{c})(\textbf{let }() = \mathsf{wait}\ \mathsf{c}\textbf{ in }t) \mid \mathsf{c}\\
(\textsc{Lift+Wait}) \rightsquigarrow\ & \textbf{let }() = ()\textbf{ in }t\\
(\textsc{LiftV}) \rightsquigarrow\ & t
\end{aligned}$$

Thus, $j(i(t)) = t : \tau$ as required.

In the opposite direction of the isomorphism we then have, $h : (\tau^{-1})^{-1}$ with

$$\begin{aligned}
i(j(h)) &= (\lambda x.\lambda f.f\ x)(\textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x)\\
&= (\lambda f.f\ (\textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }x))\\
&= \textbf{let }(x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\textbf{ in let }() = \mathsf{wait}\ c\textbf{ in }(\lambda f.f\ x)
\end{aligned}$$

Similarly to the first case, we then have the inner typing:

$$\begin{aligned}
\lambda x.\mathsf{send}\ c\ x &: \tau \multimap \mathsf{Chan}(\mathsf{end}_!)\\
h(\lambda x.\mathsf{send}\ c\ x) &: \mathsf{Chan}(\mathsf{end}_!)\\
\mathsf{fork}(h(\lambda x.\mathsf{send}\ c\ x)) &: \mathsf{Chan}(?\tau.\mathsf{end}_?)\\
\mathsf{recv}(\mathsf{fork}(h(\lambda x.\mathsf{send}\ c\ x))) &: \tau \otimes \mathsf{Chan}(\mathsf{end}_?)
\end{aligned}$$

So, again, $(x, c) : \tau \otimes \mathsf{Chan}(\mathsf{end}_?)$.

Applying the global configuration semantics of GV [30, Figure 4], we get:

$$\begin{aligned} & \textbf{let } (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \textbf{in let}\ () = \mathsf{wait}\ c\ \textbf{in}\ (\lambda f.f\ x) \\ (\textsc{Lift+Fork}) \leadsto\ & (\nu \mathsf{c})\,(\textbf{let } (x, c) = \mathsf{recv}\ \mathsf{c}\ \textbf{in let}\ () = \mathsf{wait}\ c\ \textbf{in}\ (\lambda f.f\ x) \mid h\ (\lambda x.\mathsf{send}\ \mathsf{c}\ x)) \end{aligned}$$

Recall that $h : (\tau \multimap 1) \multimap 1$ therefore we know that h must necessarily use the input parameter, applying it to some $t : \tau$, therefore after some reduction in $h\ (\lambda x.\mathsf{send}\ \mathsf{c}\ x)$ we get some $\textbf{let } c' = \mathsf{send}\ \mathsf{c}\ t\ \textbf{in}\ h' : ()$ and some configuration C in the case that evaluating to this point had some other communication effects. Note that c' is not in the free variables of h' since the session typing tells us it is unused, i.e. $c' : \mathsf{Chan}(\mathsf{end}_!)$.

Then we get the continuing reduction sequence:

$$\begin{aligned} (\textsc{LiftV}*) \leadsto^*\ & (\nu \mathsf{c})\,(\textbf{let } (x, c) = \mathsf{recv}\ \mathsf{c}\ \textbf{in let}() = \mathsf{wait}\ c\,\textbf{in}(\lambda f.f\ x) \mid \textbf{let } c' = \mathsf{send}\ \mathsf{c}\ t\ \textbf{in}\ h' \mid C) \\ (\textsc{Lift+Send}) \leadsto\ & (\nu \mathsf{c})\,(\textbf{let } (x, c) = (t, \mathsf{c})\ \textbf{in let}() = \mathsf{wait}\ c\,\textbf{in}(\lambda f.f\ x) \mid \textbf{let } c' = \mathsf{c}\ \textbf{in}\ h' \mid C) \\ (\textsc{LiftV}) \leadsto\ & (\nu \mathsf{c})\,(\textbf{let } () = \mathsf{wait}\ \mathsf{c}\ \textbf{in}\ (\lambda f.f\ t) \mid \textbf{let } c' = \mathsf{c}\ \textbf{in}\ h' \mid C) \\ (\textsc{Lift+Wait}) \leadsto\ & (\lambda f.f\ t) \mid h' \mid C \end{aligned}$$

The result is a term that behaves like the original h; applying the term t from inside h to the continuation to f results in a configuration C and has some remaining reduction to do as h'. Thus $i(j(h)) = h : (\tau \multimap 1) \multimap 1$ as required.

Compiling Volatile Correctly in Java

Shuyang Liu
University of California, Los Angeles, CA, USA

John Bender
Sandia National Laboratories, Albuquerque, NM, USA

Jens Palsberg
University of California, Los Angeles, CA, USA

Abstract

The compilation scheme for Volatile accesses in the OpenJDK 9 HotSpot Java Virtual Machine has a major problem that persists despite a recent bug report and a long discussion. One of the suggested fixes is to let Java compile Volatile accesses in the same way as C/C++11. However, we show that this approach is invalid for Java. Indeed, we show a set of optimizations that is valid for C/C++11 but invalid for Java, while the compilation scheme is similar. We prove the correctness of the compilation scheme to Power and x86 and a suite of valid optimizations in Java. Our proofs are based on a language model that we validate by proving key properties such as the DRF-SC theorem and by running litmus tests via our implementation of Java in Herd7.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases formal semantics, concurrency, compilation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.6

Related Version *Full Version*: `http://compilers.cs.ucla.edu/papers/compiling-volatile.pdf`

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.3`

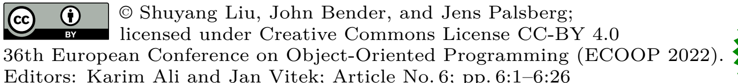
Funding This material is based upon work supported by the National Science Foundation under Grant No. 1815496.

Acknowledgements We thank Doug Lea for the helpful insights on the Java language semantics and compilers; we thank Jade Alglave for her precious and detailed help on implementing Java architecture for Herd; we thank Ori Lahav, Anton Podkopaev and Viktor Vafeiadis for initially pointing out the issue of the Java Access Modes model; we thank all the reviewers of ECOOP'22 for their insightful feedback.

1 Introduction

In OpenJDK 9, the Java programming language introduced the VarHandle API with Access Modes to provide a standard set of operations that gives clear semantics to programs with shared object fields. Among the four available Access Modes (which we will explain in Section 3 in detail), programmers are allowed to use Volatile mode to ensure the consistency of updates on shared variables. Conceptually, the set of Volatile mode accesses in a program is totally ordered [9]. If all of the accesses in a program are in Volatile mode, then the program should only have sequentially consistent executions since all accesses in that program are totally ordered.

Sadly, this basic property of Volatile mode does not hold under the current implementation of the Java compiler in OpenJDK 9 HotSpot JVM. That is, marking all accesses as Volatile in a Java program can still result in behaviors that are not sequentially consistent when compiling to Power [14]. In particular, the C1 and the C2 compilers in HotSpot do not


© Shuyang Liu, John Bender, and Jens Palsberg;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 6; pp. 6:1–6:26

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

provide enough synchronization between a Volatile read and a Volatile write when compiling to the Power architecture. While we leave the details of their respective compilation schemes to Section 2, when a program includes a sequence of a Volatile read followed by a Volatile write, there is no `hwsync` instruction inserted in-between. Without the `hwsync`, it is possible for threads to disagree on the orders in which instructions are executed. As a consequence, the compilation schemes can still cause violations of sequential consistency in programs with all accesses marked Volatile. We have contacted the maintainers of the OpenJDK about this issue and a bug report has been filed [17].

One solution is to add the missing `hwsync` instruction to restore sequential consistency for Volatile. The resulting compilation scheme is similar to C/C++11 [4], which leads one to wonder whether Java compilers can simply handle Access Modes the same way as C/C++11 compilers handle atomic memory orders. However, there are significant differences in the semantics of Volatile access mode and the `seq_cst` memory order, which leads to differences in the valid compiler transformations applied to them respectively. In contrast to C/C++11 [6], Java does not allow certain compiler transformations to be applied to Volatile accesses. For example, register promotion cannot be applied to memory locations with Volatile accesses in Java while it can be applied in C/C++11. The differences provide Java programmers stronger synchronization guarantees and a more intuitive reasoning process: Volatile accesses (1) are equivalent to inserting `fullFence()`s, and (2) will not be optimized by the compiler in unexpected ways. We provide a detailed comparison along with soundness proofs and examples in Section 5.

While the change to the compilation scheme appears to be simple, the work of verifying its soundness is challenging. First, the formal language model JAM (hereafter JAM_{19}) [3] exhibits the same issue as the HotSpot compilers. That is, it cannot guarantee sequential consistency for programs with all accesses marked Volatile. Therefore, we revise the language model to fix this issue. To ensure the change to the model is valid we formally verify its key properties, such as the standard DRF-SC theorem, and leverage a set of empirical litmus tests via our implementation of Java in Herd7 [1] that keeps the model valid. We call the revised model JAM_{21} to distinguish from the original version. Second, the language model defines the semantics of `fullFence()` with a total order. However, many target-level architectures such as the Power memory model [14] only specify a partial observable order among their synchronization mechanisms (fence cumulativity). Therefore, we develop an intermediate language model, JAM'_{21}, to bridge JAM_{21} with the target level models. We show that JAM'_{21} yields the same observable program executions as JAM_{21} but does not specify a total order among `fullFence()`s, which simplifies the proof for compilation correctness.

1.1 Outline

The rest of the paper is structured as follows. Section 2 explains the bug in the current Java compiler to Power with an example. In Section 3, we explain the formal model that we use in this paper. Section 4 provides a correctness proof for our proposed compilation scheme to Power. Section 5 presents a set of program transformations that are valid/invalid for Java and a comparison with C/C++11. We include a discussion on expected performance impact in Section 6. Section 7 details some recent related work and finally, Section 8 concludes the paper.

1.2 Supplementary Material

The proofs of the theorems appear in this paper are available in the appendices (which are available in the full version of the paper). The following are also available as artifact of this paper at `https://github.com/ShuyangLiu/ECOOP22-Supplementary-Material`.

- The extended Herd7 tool suite with the Java architecture.
- The litmus tests that appear in this paper.
- The Coq proofs for some of the theorems in this paper.

2 The Problem of Compiling Volatile and How to Fix it

In this section we use an example to demonstrate that the approach implemented by the HotSpot JVM compilers does not provide sequentially consistent semantics even when all accesses use Volatile mode.

Consider the volatile-non-sc.4 example shown as an execution in Fig.1. In this example, there are four concurrent threads (`P1`, `P2`, `P3`, and `P4`) accessing two shared integer variables (x and y). The notation `Wx = 1` means "writing to variable x with value 1". The notation `Rx = 0` means "reading from variable x and the value returned is 0". In addition, each variable is initialized to 0 at the beginning before the threads start execution. The small superscript on each memory access denotes the access mode that the access uses. For example, $\texttt{Rx}^{\texttt{v}}$ means "reading with Volatile mode".

If all of the read and write accesses in this program use Volatile mode, would the reads ever return the values that are specified in the figure?

According to the specification [9], the program must exhibit sequentially consistent behavior because all accesses are marked Volatile:

> *"When all accesses use Volatile mode, program execution is sequentially consistent, in which case, for two Volatile mode accesses A and B, it must be that A precedes execution of B, or vice versa."*

Therefore, we are interested in whether the example in Fig. 1 is sequentially consistent. Sequential consistency, as first defined by [7], requires a total sequential order that preserves program order and the values returned by the reads are compatible with this total order. Following the definition, the execution in Fig. 1 does not satisfy sequential consistency. To see this, we demonstrate a contradiction under the guarantees of sequential consistency. Consider the following order constraints:

1. By program order, we know that (a) occurs before (b).
2. Since the value (b) gets is the initial value 0, it must occur before (c) writes to the location y.
3. Then, (d) reads the value written by (c), so (c) occurs before (d).
4. By program order, (d) occurs before (e).
5. Now, looking at `P4`, we know that the value of x changed from 1 to 2. Therefore, we can infer that (e) occurs before (a) since (e) is the only write to x with a value of 1 and (a) is the only write to x with a value of 2.

In this execution, we find a cycle: (a) $\longrightarrow$ (b) $\longrightarrow$ (c) $\longrightarrow$ (d) $\longrightarrow$ (e) $\longrightarrow$ (a) which appears in Fig. 1 with the "occurs before" relation represented as edges in the execution graph. Sequential consistency requires an irreflexive total order among all instructions. Therefore, the chain formed by the total order should be acyclic, i.e., a valid execution should not exhibit any cycle in its graph. Thus, this execution is inconsistent under sequential consistency and should be forbidden.

```
                 Wx = 0     Wy = 0

       P1             P2             P3              P4

(a)  Wx^V = 2   (c) Wy^V = 1 ->  (d) Ry^V = 1   (f) Rx^V = 1

(b)  Ry^V = 0                    (e) Wx^V = 1   (g) Rx^V = 2
```

Figure 1 `volatile-non-sc.4` under the sequential consistency model, Forbidden.

```
                 Wx = 0     Wy = 0

       P1             P2             P3              P4

(a)  Wx = 2      (c) Wy = 1     (d) Ry = 1      (f) Rx = 1
(B1) hwsync                     (B2) lwsync     (B3) hwsync
(b)  Ry = 0                     (e) Wx = 1      (g) Rx = 2
```

Figure 2 volatile-non-sc.4.ppc translated to Power by HotSpot C1, Allowed.

However, despite the promise of sequential consistency given by the source-level Volatile semantics, the compilation scheme found in the Java compilers for Power allows the example execution in Fig. 1. To see this, we present the compilation scheme from the C1 compiler which is the more conservative of HotSpot's two compilers. We then give a Power-consistent execution graph corresponding to the example in Fig. 1.

The Power architecture adopts a relaxed memory model and provides fence instructions to recover sequential consistency. Two main types of fence instructions, the stronger fence `hwsync` and the weaker fence `lwsync`, are usually used by the compilers to enforce synchronization guarantees. Using `lwsync` usually gives better performance but the synchronization guarantee of `lwsync` is weaker than `hwsync`. In particular, while both fence instructions carries a set of writes (Group A writes) when propagating to another thread, `lwsync` does not require an acknowledgement to continue executing the instructions after it. On the other hand, a `hwsync` requires an acknowledgment marking that it (along with its Group A writes) has propagated to all threads before proceeding to the next instruction.

The compilation to Power for Volatile accesses on C1 is the following [1]:

$$R^V \rightsquigarrow \texttt{hwsync ; lwz ; lwsync}$$
$$W^V \rightsquigarrow \texttt{lwsync ; stw ; hwsync}$$

A Volatile read is compiled to a `hwsync` instruction followed by a load instruction and a `lwsync` instruction; a Volatile write is compiled to a `lwsync` instruction followed by a store instruction and a `hwsync` instruction.

Fig. 2 shows the example from Fig. 1 according the compilation scheme in the C1 compiler[2].

[1] This compilation scheme was found in the OpenJDK 13 HotSpot compiler and it follows from a previously inaccurate description in the documentation [9] regarding the semantics of Volatile accesses. We have contacted the author and the documentation has been corrected in the latest version while the compiler bug (although reported) is still not fixed at the time of writing.

[2] The C2 compiler yields a slightly different compilation scheme for Volatile reads: Instead of inserting a `lwsync` fence after the load instruction, it emits a control dependency followed by an `isync` instruction, which we denote as `ctrlisync`. But in this example, the resulting execution graph is effectively the same

```
                    Wx = 0      Wy = 0

      P1              P2              P3                P4

(a)  Wx = 2    (c)  Wy = 1    (d)  Ry = 1      (f)  Rx = 1
(B1) hwsync                   (B2) hwsync      (B3) hwsync
(b)  Ry = 0                   (e)  Wx = 1      (g)  Rx = 2
```

Figure 3 volatile-non-sc.4.ppc translated to Power using the revised compilation scheme, Forbidden.

The Power memory model [14] allows the behavior annotated in Fig. 2. The full trace of the execution can be found in the full version of this paper. Here we give a brief explanation. First note that a write operation is split into multiple steps and can be propagated to foreign threads in different orders if not properly synchronized. Furthermore, the `lwsync` in `P3` is not sufficient in this case. In particular, the `lwsync` does not require an acknowledgement before proceeding to the next instructions and it only requires (c) `Wy = 1` to be propagated when itself needs to be propagated to the thread (the cumulativity of `lwsync`). Since `P4` needs to read from (e) `Wx = 1`, which is subsequent to (B2), (B2) needs to be propagated to `P4` before (e) `Wx = 1` is propagated to `P4`. The propagation of (B2) `lwsync` makes sure that (c) `Wy = 1` is propagated to `P4` before it can read x (even though it doesn't really need to read the value of y). On the other hand, `P1` does not have any instructions reads from an instruction of `P3` that comes after (in program order) (B2). Therefore, it does not require (c) and (B2) to be propagated to it when it executes (b). As a result, (c) can be propagated to `P1` long after reaching `P3` and hence letting `P3` and `P1` have different views of the memory during the execution. When `P1` tries to read the value of y, it can only get an initial value of 0 since the newer value has not been propagated to `P1` yet. Consequently, this non-SC execution is allowed (consistent) under the Power memory model, despite that the semantics of the "all-Volatile" source program requires it to be forbidden.

The solution to fix this issue is quite straightforward. Instead of letting Volatile read be translated using "leading fence" while Volatile write be translated using "trailing fence", they should both use the same fence inserting strategy (both leading fence or both trailing fence).[3] Therefore, the correct compiler scheme for Volatile should be:

$$R^{V} \rightsquigarrow \texttt{hwsync ; lwz ; lwsync}$$
$$W^{V} \rightsquigarrow \texttt{hwsync ; stw}$$

With the revised compilation scheme we can demonstrate that the example of Fig. 1 is forbidden in accordance with the required SC semantics. The resulting execution graph is shown in Fig. 3. While most of this example matches Fig. 2, (B2) now is a `hwsync` instruction. As an effect of this change, (B2) is now required to be propagated to every thread and get

as C1's because the effect of `ctrlisync` is subsumed into the `lwsync` or the `hwsync` instruction that it follows. In addition, we have simplified the compiled code (such as eliminating the fence instructions at the beginning or end of the threads and merging consecutive fence instructions) without changing its semantics for clarity here.

[3] Here we choose to show the leading fence strategy for simplicity. However, the trailing fence strategy is symmetric to leading fence and the same correctness proof works for both conventions given it's used consistently (more details can be found in Section 4.1). In practice, it is usually preferable to use trailing fence strategy for better performance.

acknowledged before start executing (e). As a result, at the time when (c) is propagated to P4 (as a result of the cumulative effect of (B2) just like in Section. 2), it must also have propagated to P1 due to the acknowledgement required by the hwsync at (B2). Therefore, it becomes impossible for (b) to read the value 0 because Power requires reads to always read from the latest value that has been propagated to the thread. That is, this execution is now forbidden by Power, aligning with the sequentially consistent semantics promised by the Java Volatile mode. Note that the reasoning is the same if we use a "trailing fence" scheme. The key is to deploy a fence insertion strategy such that there is a hwsync fence inserted between every pair of Volatile accesses.

Interestingly, we found similar compilation schemes applied to other architectures in HotSpot as well. This is not an accident. The source of this compiling behavior stems from the IR phase of the compiler. At the IR (called the Ideal Graph IR in HotSpot) level, a Volatile read is translated to a `fullFence()` followed by an Acquire read; a Volatile write is translated to a Release write followed by a `fullFence()`. Then each compiler back end translates the code further using the corresponding template file that maps the IR to specific architecture instructions. In the case of Power, a `fullFence()` is mapped to the hwsync instruction and Release-Acquire accesses are implemented using the lwsync instruction. While the example we provide here focuses on the compilation to Power, the more fundamental issue here is a lack of `fullFence()` between a Volatile read and a Volatile write at the IR encoding level. JAM_{19} aligns with this encoding when specifying the semantics of Volatile memory operations. As a result, JAM_{19} also exhibits the same problem. That is, when all memory accesses are Volatile, JAM_{19} does not guarantee sequential consistency.

3 Formal Model

In this section we present the revised model JAM_{21}, which we use as our theoretical foundation for proving compiler correctness in the rest of the paper. We begin by introducing the basic syntax (Section 3.1) used in the rest of the paper. Then we give the formal definition of JAM_{21} in Section 3.2.

3.1 Basic Syntax

We adopt the syntax of [3] and the `cat` language [1] in addition to some utility functions.

Given a program $P \in \mathbb{P}$, there is a set of executions (run-time traces) associated with P. We call the executions *histories* of P and use H to denote a single history. Each execution history consists of sets of memory access events specified by P. In particular:

- $H.\texttt{E}$ denotes the whole set of memory events of H.
- $H.\texttt{F}$ denotes the whole set of fence events of H.
- $H.\texttt{IW}$ denotes the set of initialization writes of H.
- $H.\texttt{FW}$ denotes the set of final writes of H
- $H.\texttt{W}$ denotes the set of write events in H.
- $H.\texttt{R}$ denotes the set of read events in H.
- $H.\texttt{RMW}$ denotes the set of read-modify-write events in H.

Note that we treat each RMW events as a single event and $H.\texttt{RMW} \subseteq H.\texttt{W}$ and $H.\texttt{RMW} \subseteq H.\texttt{R}$. In addition, for RMW operations such as *compare-and-swap* (CAS), we assume the operation is on its success comparison path. They are sometimes implemented using LL/SC instructions on hardware, which cannot guarantee atomicity if the comparison fails. We assume each write event to the same memory location has an unique value for simplicity.

For each memory event i, we define the following utility functions to extract memory event attributes:

- $H.AccessMode(i)$ returns the Access Mode of event i in H.
- $H.val(i)$ returns the value of event i in H.
- $H.loc(i)$ returns the shared memory location of event i in H.
- $H.Tid(i)$ returns the thread identifier of which i is executed from

Finally, we use the symbol $\mathbb{H}$ to denote the set of all execution histories.

The memory events of each H are related by order relations.

- The program order (po) is a partial order relation ($\texttt{po} \subseteq H.\texttt{E} \times H.\texttt{E}$) specified by P. We use the notation $i_1 \xrightarrow{\texttt{po}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle$ related by po and $H.\texttt{po}$ to denote the set of all pairs relates by po in H.
- The reads-from (rf) order is a partial order relation ($\texttt{rf} \subseteq H.\texttt{W} \times H.\texttt{R}$). For each read event i_2, there exists a unique write event i_1 such that $H.val(i_1) = H.val(i_2)$ and $H.loc(i_1) = H.loc(i_2)$. We use the notation $i_1 \xrightarrow{\texttt{rf}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle$ related by rf and $H.\texttt{rf}$ to denote the set of all pairs relates by rf in H.
- Model-Specific relations. There are sets of relations that are specifically defined by the memory model. They are derived from the event attributes, po, and rf using the semantic rules of the memory model. We will detail them in the next few sections. We use the notation $i_1 \xrightarrow{\texttt{R}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle \in H.\texttt{R}$.

We also use operations on relations: given relations $\texttt{R}_1$ and $\texttt{R}_2$, we use composition $\texttt{R}_1;\texttt{R}_2$, union $\texttt{R}_1|\texttt{R}_2$, intersection $\texttt{R}_1\&\texttt{R}_2$, complement $\sim\texttt{R}_1$, transitive closure $\texttt{R}_1^+$, and inversion $\texttt{R}_1^{-1}$.

We may present an execution history H as a graph. An execution graph consists of a set of nodes labeled with unique identifiers, and a set of labeled edges. Each labeled node refers to an executed memory access.

Lastly, we use the notation $\text{acyclic}(\xrightarrow{\texttt{R}})$ to denote that R is acyclic in the execution history.

3.2 The JAM_{21} Model

In this section, we present the JAM_{21} model. The full definition of the relations in JAM_{21} can be found in the full version of this paper. We explain several excerpts of the formal model.

There are five available access modes in JAM_{21}: Plain mode, Opaque mode, Release mode, Acquire mode, and Volatile mode. The synchronization effect of the access modes are partially ordered using $\sqsubseteq$:

$$\text{Plain} \sqsubseteq \text{Opaque} \sqsubseteq \{\text{Release}, \text{Acquire}\} \sqsubseteq \text{Volatile}.$$

3.2.1 Visibility

At the center of JAM_{21} is the notion of *visibility orders* (vo). The most basic form of visibility, vo includes the reads-from (rf) relation. Intuitively, a read has certainly seen the effects of the write it takes its value from. Otherwise, visibility comes from synchronization[4]. Both

[4] Here, we use the high-level term "synchronization" for any memory consistency guarantee among instructions. We noticed that the usage of this term might differ outside of this paper. Therefore, we try to avoid using this term ambiguously to avoid confusion.

Volatile (V) and Release(REL)-Acquire(ACQ), (RA as the union) accesses provide synchronization and thus visibility. Note that Volatile accesses are also included in the set of accesses that are considered Release-Acquire by the model. Further, vo can be derived from ra or svo orders, which captures the synchronization effects of Release-Acquire memory events or fences, spush or volint orders, which capture the synchronization effects of Volatile memory events or fullFence()s. In addition, the pushto order is trace order (to) restricted to the domain of spush and volint. Composing pushto with spush or volint emulates the cross-thread total order among fullFence()s, which is also part of the vo order. Finally, po to the same location is also included as part of the vo definition.

$$\begin{aligned}
\texttt{ra} &\triangleq \texttt{po ; [REL | V] | [ACQ | V] ; po}\\
\texttt{svo} &\triangleq \texttt{po ; [F \& REL] ; po ; [W] | [R] ; po ; [F \& ACQ] ; po}\\
\texttt{spush} &\triangleq \texttt{po ; [F \& V] ; po}\\
\texttt{volint} &\triangleq \texttt{[V] ; po ; [V]}\\
\texttt{vvo} &\triangleq \texttt{rf | svo | ra | spush | volint | pushto ; (spush | volint)}\\
\texttt{vo} &\triangleq \texttt{vvo}^{+}\texttt{ | po-loc}
\end{aligned}$$

Note that the definition of volint has been corrected from JAM_{19} to ensure sequential consistency for Volatile.

3.2.2 Coherence

The coherence order, co-jom, is an order among writes to the same location. Coherence order edges can be derived using the vo order and the po order among memory accesses.

$$\begin{aligned}
\texttt{WWco(rel)} &\triangleq \{\langle i_1, i_2\rangle \mid \langle i_1, i_2\rangle \in H.\texttt{rel} \land i_1, i_2 \in H.\texttt{W} \land H.loc(i_1) = H.loc(i_2) \land i_1 \neq i_2\}\\
\texttt{coww} &\triangleq \texttt{WWco(vo)}\\
\texttt{cowr} &\triangleq \texttt{WWco(vo ; rf}^{-1}\texttt{)}\\
\texttt{corw} &\triangleq \texttt{WWco(vo ; po)}\\
\texttt{corr} &\triangleq \texttt{[O | RA | V] ; WWco(rf ; po ; rf}^{-1}\texttt{) ; [O | RA | V]}\\
\texttt{co-jom} &\triangleq \texttt{coww | cowr | corw | corr}
\end{aligned}$$

Note that co-jom is different from the definition of co in other memory models such as Power and x86-TSO. Instead of enumerating all possible total coherence order to check the consistency of a given execution history, JAM_{21} derives coherence order co-jom among memory events from their known relations. Therefore, co-jom is a partial order among writes to the same location in JAM_{21}. We use the notation $i_1 \xrightarrow{\texttt{co-jom}} i_2$ to denote the pair of events $\langle i_1, i_2\rangle$ related by co-jom and $H.$co-jom to denote the set of all pairs relates by co-jom in H. We use the simpler name co to denote co-jom when the context is clear.

In addition, different from JAM_{19}, Plain mode reads to the same location ordered by po can be reordered by compiler and therefore cannot be used to derive co-jom order.

3.2.3 Execution Consistency

Axiomatic models define program semantics as the set of allowed executions. We adopt the same definition of *candidate execution* from [1].

▶ **Definition 1** (Consistent Candidate Execution). *Given a program P and a memory model M, an execution history H is a **M-consistent candidate execution of P** if and only if:*
- *H is a candidate execution of P (specified by the architecture of the programming language of which P is written in).*
- *H is M-consistent.*

We denote the set of all M-consistent candidate executions of P by $Histories_M(P)$.

We now have all the definitions needed to define execution consistency under JAM_{21}.

▶ **Definition 2** (JAM_{21}-consistency). *An execution history H is JAM_{21}-**consistent** if it is trace coherent and satisfies the following two requirements:*
1. *NO-THIN-AIR: po / rf is acyclic.* acyclic($\xrightarrow{\texttt{po}|\texttt{rf}}$)
2. *COHERENCE.: co-jom is acyclic,* acyclic($\xrightarrow{\texttt{co-jom}}$)

*We say such an execution history H is **allowed** by JAM_{21}. Otherwise, it is **forbidden**.*

For the JAM_{21} model, we use $Histories_{JAM_{21}}(P)$ to denote the set of all JAM_{21}-consistent execution histories of P.

JAM_{21} satisfies a set of properties such as the DRF-SC Theorem. We show the theorems and the proofs in the full version of this paper.

3.2.4 Validation with Litmus Tests

The experimental validation of the JAM_{21} model includes two parts.

First, we implement the Java *architecture* in Herd7. Herd7 [1] was developed to simulate program executions with user-defined memory models. An *architecture* in Herd7 provides the parser for litmus tests written in the language corresponding to the architecture and an operational semantics of the instructions that appear in litmus tests. Herd7 uses the parser and the instruction semantics from the architecture to form an internal representation of the input litmus test and generate the set of all possible executions. Then, Herd7 checks the consistency of the executions using memory models written in the `cat` language. As of today, several mainstream architectures, such as C/C++11 [6], x86 [15], ARM [2], and Power [14], have been implemented and included in Herd7's official repository. Unfortunately, Java is not. JAM_{19} [3] validated its formalization by mapping memory events to other architectures' events that exists in the Herd7 repository and run the litmus tests in the architecture's language. The mapping roughly captures part of the compilation scheme but it is neither complete nor proven sound. For example, in its mapping to ARMv8, Volatile accesses are ignored and not mapped to any memory event. Hence this approach is invalid and the results cannot be trusted though they show intentions on how JAM_{19} was expected to behave. Therefore, we extend the Herd7 tool suite with the Java architecture and translate the set of litmus tests used for testing JAM_{19} to Java[5]. A detailed description of each supported instruction is shown in the full version of this paper.

Second, we validate the JAM_{21} model using the Java translation of the set of litmus tests that was originally used to validate JAM_{19} and compare their outcomes. The results are mostly the same as the results from JAM_{19} except for three cases that are relevant to the inconsistency issue discussed earlier in this paper because we wish to fix the issue while keeping other parts of the model unchanged. The three exceptions reveal another

[5] Note that not all tests are translatable. For example, for the cases that test address dependencies, there is no corresponding Java version since the notion of address dependency does not exist in Java. We drop a small set of litmus tests due to this reason.

aspect of the change, accommodating both the leading fence convention and the trailing fence convention, whereas JAM_{19} forced the compiler to choose a particular (problematic) convention. Since the compiler is free to choose either convention, a full synchronisation is only guaranteed to appear between a pair of Volatile accesses. In effect, certain executions that was forbidden by JAM_{19} are allowed by JAM_{21} since it is no longer guaranteed that Volatile writes are *followed* by a full synchronisation and Volatile reads are *prepended* with a full synchronisation. In addition, we have added new litmus tests for showing the change in the semantics of Volatile, volatile-non-sc.4 and volatile-non-sc.5. While JAM_{19} allows the non-sequentially consistent behavior, JAM_{21} correctly forbids them. We further translated the examples to Power using the problematic compilation scheme, volatile-non-sc.4.ppc and volatile-non-sc.5.ppc, and the tests are indeed allowed by the Power memory model. Please see the full version of this paper for a detailed report.

4 Compilation Correctness to Power

In this section, we show that the revised compilation scheme for Power is correct with respect to the Power memory model [14]. We use an intermediate model for the Java Access Modes that is observationally equivalent to JAM_{21}, which we call JAM'_{21}. We include the detailed definition of JAM'_{21} and the proofs for their observational equivalence in the full version of this paper. We use JAM'_{21} to prove that the revised compilation scheme to Power is correct.

4.1 The Power Memory Model

We use the Power memory model defined in Herd7 [1], which consists of the following basic order definitions (Please see the full version of this paper for the full semantics):

- po and rf follows the same definitions as in JAM_{21} (as described in Section. 3).
- co is the union of total orders among writes to the same location. Additionally, if i_1 and i_2 are events on different threads and $i_1 \xrightarrow{co} i_2$, then $i_1 \xrightarrow{coe} i_2$.
- ctrl is the control dependency between memory accesses.
- ppo is the set of preserved program orders. The detailed definition can be found in the full version of this paper.
- chapo $\triangleq$ rfe | fre | coe | (fre ; rfe) | (coe ; rfe)
- com $\triangleq$ rf | fr | co
- po-loc is a subset of po that relates accesses to the same locations.
- rmw relates the read and the write access from the same RMW memory event.
- hb $\triangleq$ ppo | (sync | lwsync) | rfe
- propbase $\triangleq$ ((sync | lwsync) | (rfe ; (sync | lwsync))) ; hb*
- prop $\triangleq$ propbase & (W ∗ W) | (chapo? ; propbase* ; sync ; hb*)
- Additional order definitions can be found in the full version of this paper.

▶ **Definition 3** (Power Consistency). *An execution history H is Power-**consistent** if it is trace coherent and satisfies the following six requirements:*

1. *SC-per-location:* po-loc | com *is acyclic.*
2. *Atomicity:* rmw & (fre ; coe) *is empty.*
3. *No-Thin-Air:* hb *is acyclic.*
4. *Propagation:* (co | prop) *is acyclic.*
5. *Observation:* fre;prop;hb* *is irreflexive.*
6. *SCXX:* co | (po & (X ∗ X)) *is acyclic (where* X *denotes atomic accesses)*

*We say such an execution history H is **allowed** by Power. Otherwise, it is **forbidden**.*

```
getOpaque() ⇝ lwz ; cmp ; bc
setOpaque() ⇝ stw
getAcquire() ⇝ lwz ; lwsync
setRelease() ⇝ lwsync ; stw
getVolatile() ⇝ hwsync ; lwz ; lwsync
(Or getVolatile() ⇝ lwz ; hwsync for trailing fence convention)
setVolatile() ⇝ hwsync ; stw
(Or setVolatile() ⇝ lwsync ; stw ; hwsync for trailing fence convention)
AcquireFence() ⇝ lwsync
ReleaseFence() ⇝ lwsync
fullFence() ⇝ hwsync
getAndAdd() ⇝ hwsync ; _1: ldarx ; add ; stdcx. ; bne _1 ; lwsync
(Or getAndAdd() ⇝ lwsync ; _1: ldarx ; add ; stdcx. ; bne _1 ; hwsync for trailing
                                                                  fence convention)
getAndAddAcquire() ⇝ _1: ldarx ; add ; stdcx. ; bne _1 ; lwsync
getAndAddRelease() ⇝ lwsync ; _1: ldarx ; add ; stdcx. ; bne _1
```

Figure 4 Compilation to Power.

4.2 Compilation Scheme

We use the compilation scheme in Fig. 4. Note that this is slightly different from the compilation scheme found in OpenJDK HotSpot compiler in that each Opaque mode read is translated to a load instruction followed by a conditional branch. This enables us to ensure the No-Thin-Air property as it is not guaranteed in the Power memory model. The problem of Out-of-Thin-Air in axiomatic models has been an active research area for a long time and there exists various ways to use weaker compilation schemes while still ruling out thin-air reads. However, it is out of the scope of this paper and here we adopt the stronger scheme for Opaque mode to simplify the proofs. Additionally, we fix the compilation scheme for Volatile as suggested in Section 2. Note that both leading fence and trailing fence conventions ensure a `hwsync` instruction is inserted between each pair of Volatile mode accesses as long as they are used consistently (use the same convention for Volatile writes and reads). Therefore, the proof for the trailing fence convention can be carried out in a very similar way as the proof for the leading fence convention.

We start our proof by defining a *CompilesTo* relation over execution histories that relates source level executions to target level executions. Intuitively, the process of compilation can be seen as a transformation function on executions from source level to target level. With the *CompilesTo* relation, we can characterize a subset of target level executions that are constructed particularly through the compilation (following a given compilation scheme) from the source level. Note that at this step we do not check whether the resulting execution is consistent under the target level memory model, since the consistency of an execution is checked after the execution is constructed in axiomatic memory models.

▶ **Definition 4** (Compilation of an Execution). *We define the "CompilesTo" relation $\rightsquigarrow \subseteq \mathbb{H} \times \mathbb{H}$ for the compilation from Java to Power as the following: Given a Java program P_{src}, let P_{tgt} be the target-level program compiled from P_{src} using the compilation scheme in Fig. 4 (using the leading fence convention). Let H_{src} be a candidate execution history of P_{src} and H_{tgt} be a candidate execution history of P_{tgt}. We say $H_{src} \rightsquigarrow H_{tgt}$ if:*

- $H_{tgt}.IW = H_{src}.IW$

- $H_{tgt}.FW = H_{src}.FW$
- $H_{tgt}.E = H_{src}.E$
- $H_{tgt}.\texttt{rf} = H_{src}.\texttt{rf}$
- $H_{tgt}.\texttt{po} = H_{src}.\texttt{po}$
- $H_{tgt}.\texttt{co} \subseteq H_{src}.to$
- *If* $i_1 \in H_{src}.\texttt{E}$, $i_{rmw} \in H_{src}.\texttt{RMW}$ *and* $i_{rmw} \xrightarrow{\texttt{po}} i_1$, *then* $i_{rmw} \xrightarrow{\texttt{ctrl}} i_1$ *in* H_{tgt}
- *If* $i_R^{\sqsupseteq O} \in H_{src}.\texttt{R}$, $i_1 \in H_{src}.\texttt{E}$ *and* $i_R^{\sqsupseteq O} \xrightarrow{\texttt{po}} i_1$, *then* $i_R \xrightarrow{\texttt{ctrl}} i_1$ *in* H_{tgt}
- *If* $i_1, i_2 \in H_{src}.\texttt{E}$ *and* $i_1 \xrightarrow{\texttt{push}} i_2$, *then* $i_1 \xrightarrow{\texttt{sync}} i_2$ *for* $i_1, i_2 \in H_{tgt}.\texttt{E}$
- *If* $i_1, i_2 \in H_{src}.\texttt{E}$ *and* $i_1 \xrightarrow{\texttt{ra}} i_2$, *then* $i_1 \xrightarrow{\texttt{lwsync}} i_2$ *for* $i_1, i_2 \in H_{tgt}.\texttt{E}$

Once we have the source level and target level execution histories, we use the memory model to check for consistency. A correct compilation, intuitively, should not introduce any new program behavior. In this context, it means there should not be any execution H_{src} that is forbidden by the source level memory model being related (by the "CompilesTo" relation) with a H_{tgt} that is allowed by the target level memory model. That is, if H_{tgt} is consistent under the target level memory model, then H_{src} should also be consistent under source level memory model. Formally, we have the following definition (recall that we use $Histories_M(P)$ to denote the set of consistent execution histories if a program P under a memory model M).

▶ **Definition 5** (Compilation Correctness). *Let P_{src} be a source program and S be a memory model that supports the source language, P_{tgt} be the target program compiled from P_{src} using a compilation scheme and T be a memory model that supports the target language. We say a compiler that compiles P_{src} to P_{tgt} is **correct** if for all $H_{tgt} \in Histories_T(P_{tgt})$ there exists a $H_{src} \in Histories_S(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

4.3 Proof of Compilation Correctness

We leverage an intermediate memory model, JAM'_{21}, to prove the compilation correctness to Power. While the complete definition of JAM'_{21} can be found in the full version of this paper, it is important to note that JAM'_{21} is *observationally equivalent* to JAM_{21}, which means they allow the same visible program behaviors given the same program. Intuitively, each consistent execution under JAM_{21} has a corresponding consistent execution under JAM'_{21} with the same set of events and the same observable value on each event. Formally, we give the following definitions for observational equivalence.

▶ **Definition 6** (Observational Equivalence of Execution Histories). *Given a program P, let H and H' be two execution histories of P. We say H and H' are **observationally equivalent** if:*

- $H.\texttt{IW} = H'.\texttt{IW}$
- $H.\texttt{FW} = H'.\texttt{FW}$
- $H.\texttt{E} = H'.\texttt{E}$
- $H.\texttt{po} = H'.\texttt{po}$
- $H.\texttt{rf} = H'.\texttt{rf}$
- $\forall i \in H.\texttt{E}, H.AccessMode(i) = H'.AccessMode(i)$

▶ **Definition 7** (Observational Equivalence of Memory Models). *Given a program P, let M_1 and M_2 be two memory models that support the architecture of the programming language that P is written in. Let $Histories_{M_1}(P)$ be the set of all M_1-consistent candidate executions of P; let $Histories_{M_2}(P)$ be the set of all M_2-consistent candidate executions of P. We say M_1 and M_2 are **observationally equivalent** if:*

- *(⇒) For all $H_1 \in Histories_{M_1}(P)$, there exists $H_2 \in Histories_{M_2}(P)$ such that H_1 is observationally equivalent to H_2.*
- *(⇐) For all $H_2 \in Histories_{M_2}(P)$, there exists $H_1 \in Histories_{M_1}(P)$ such that H_2 is observationally equivalent to H_1.*

Then we prove the compilation correctness from JAM'_{21} to Power.

▶ **Lemma 8** (JAM'_{21} to Power)**.** *Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 4 (with the leading fence convention). For all $H_{tgt} \in Histories_{Power}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM'}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

Please see the full version of this paper for the proof.

Finally, we associate JAM_{21} with JAM'_{21} through the notion of observational equivalence and prove the compilation correctness from JAM_{21} to Power.

▶ **Theorem 9** (Compilation Correctness to Power (Leading Fence Convention))**.** *The compilation from Java to Power following the compilation scheme in Fig. 4 (using the leading fence convention) is correct. That is, let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 4 (using the leading fence convention). For all $H_{tgt} \in Histories_{Power}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

Please see the full version of this paper for the proof.

▶ **Corollary 10** (Compilation Correctness to Power (Trailing Fence Convention))**.** *The compilation from Java to Power following the compilation scheme in Fig. 4 (using the trailing fence convention) is correct. That is, let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 4 (using the trailing fence convention). For all $H_{tgt} \in Histories_{Power}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

Please see the full version of this paper for the proof.

5 Compiler Transformations

One important aspect of compilers is the program transformations that they apply to the program. A correct compiler transformation should not introduce any new program behavior. While this is relatively simple for sequential programs, it can yield subtle issues when applying the same transformations to concurrent programs. A memory model's task is then to accommodate a set of common program transformations while still provide intuitive synchronization guarantees to the programmers. In Section 4 we show that Java and C/C++11 can use the same compilation scheme to Power (and x86, please see the full version of this paper). However, Java has a stronger semantics for Volatile comparing to `seq_cst` in C/C++11 and can adopt only a strict subset of the transformations that are valid for C/C++11.

In this section, we use the set of compiler transformations detailed by [6] and compare their soundness in Java with C/C++11. We provide formal proofs for the sound transformations and counter-examples for invalid transformations. We conclude this section by discussing the implications of our results.

To prove a transformation is valid, intuitively, we show that there does not exist a H_{src} of P_{src} such that it is forbidden by JAM_{21} but the corresponding H_{tgt} of P_{tgt} is allowed.

Transformation		C/C++11	Java
Strengthening	[Sec. 5.1]	✓	✓
Sequentialisation	[Sec. 5.2]	✓	✓
Reordering	[Sec. 5.3]	See Fig. 6	
Merging	[Sec. 5.4]	See Fig. 7	
Register Promotion	[Sec. 5.5]	✓	For locations that does not have Volatile access

Figure 5 Compiler Transformations in C/C++11 and Java.

▶ **Definition 11** (Valid Program Transformation). *Let P_{src} be a Java program which has a set of candidate executions, $Histories(P_{src})$. Let $T : \mathbb{H} \to \mathbb{H}$ be a program transformation and $H_{tgt} = T(H_{src})$ for each candidate execution H_{src} of P_{src}. Then we say T is **valid** under JAM_{21} if and only if for each H_{tgt}, if H_{tgt} is JAM_{21}-consistent, then H_{src} is also JAM_{21}-consistent.*

The results for Java comparing them C/C++11 [6] are summarized in Fig. 5.

5.1 Strengthening

Strengthening transforms the access mode of accesses to stronger access modes. It is supported by JAM_{21} due to the monotonicity property of the memory model. The formal theorem is the following:

▶ **Theorem 12** (Strengthening). *Let H_{tgt} an execution of P_{tgt}, which is obtained from applying Strengthening to a program P_{src}. There exists an execution H_{src} of P_{src} such that:*

- $H_{src}.E = H_{tgt}.E$
- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf}$
- $\forall i \in H_{src}.E, H_{src}.AccessMode(i) \sqsubseteq H_{tgt}.AccessMode(i)$

If H_{tgt} is JAM_{21}-consistent, then H_{src} is JAM_{21}-consistent.

Proof. By Monotonicity of JAM_{21}, all the constraints in H_{src} are preserved in the strengthened execution H_{tgt}. Therefore, if H_{tgt} is JAM_{21}-consistent, so is H_{src}. ◀

5.2 Sequentialisation

Sequentialisation transforms two concurrent accesses into accesses in a single sequential process. It is natually supported by JAM_{21} because sequentialisation does not remove any synchronization from the program.

▶ **Theorem 13** (Sequentialisation). *Let P_{src} be a Java program and P_{tgt} be a Java program obtained by performing a sequentialisation operation on a pair of accesses a and b. Let H_{tgt} be an execution of P_{tgt}. Then there exists an execution H_{src} of P_{src} such that*

- $H_{src}.\texttt{po} \cup \{\langle a, b\rangle\} = H_{tgt}.\texttt{po}$ *where* $\langle a, b\rangle \notin H_{src}.\texttt{po}$ *and* $\langle b, a\rangle \notin H_{src}.\texttt{po}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf}$
- $H_{src}.\texttt{E} = H_{tgt}.\texttt{E}$
- $H_{src}.to = H_{tgt}.to$
- $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$
- $\forall i \in H_{src}.\texttt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$

and if H_{tgt} is JAM_{21}-consistent, then H_{src} is JAM_{21}-consistent.

	$\mathtt{R}_y^{m_2}$	$\mathtt{W}_y^{m_2}$	$\mathtt{RMW}_y^{m_2}$	$\mathtt{F}^{m_2}$
$\mathtt{R}_x^{m_1}$	$m_1 \sqsubseteq$ Opaque	$m_1, m_2 \sqsubseteq$ Opaque $\wedge$ $(m_1 =$ Plain $\vee$ $m_2 =$ Plain$)$	$m_1 =$ Plain $\wedge$ $m_2 \sqsubseteq$ Acquire	$(m_1 \sqsubseteq$ Opaque $\wedge$ $m_2 =$ Release $\wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\mathtt{po}} i \Rightarrow i \notin H.\mathtt{W}) \vee (m_1 =$ Acquire $\wedge$ $m_2 =$ Acquire$) \vee (m_1 =$ Acquire $\wedge$ $m_2 =$ Release$)$
$\mathtt{W}_x^{m_1}$	$m_1 \neq$ Volatile$\vee$ $m_2 \neq$ Volatile	$m_2 \sqsubseteq$ Opaque	$m_2 \sqsubseteq$ Acquire	$(m_2 =$ Acquire$) \vee (m_2 =$ Release $\wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\mathtt{po}} i \Rightarrow i \notin H.\mathtt{W}) \vee (m_2 =$ Release $\wedge$ $\forall i, \mathtt{F}^{m_2} \xrightarrow{\mathtt{po}} i \wedge i \in H.\mathtt{W} \Rightarrow AccessMode(i) =$ Release$)$
$\mathtt{RMW}_x^{m_1}$	$m_1 \sqsubseteq$ Release	$m_1 \sqsubseteq$ Release$\wedge$ $m_2 =$ Plain	-	$(m_1 \sqsupseteq$ Acquire $\wedge$ $m_2 =$ Acquire$) \vee (m_2 =$ Release $\wedge$ $\forall i, \mathtt{F}^{m_2} \xrightarrow{\mathtt{po}} i \Rightarrow (i \in H.\mathtt{R} \vee (i \in H.\mathtt{W} \wedge AccessMode(i) =$ Release$)))$
$\mathtt{F}^{m_1}$	$(m_1 =$ Release$) \vee (m_1 =$ Acquire $\wedge$ $\forall i, i \xrightarrow{\mathtt{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R})$	$m_1 =$ Release$\wedge$ $m_2 \sqsupseteq$ Release$\vee$ $(m_1 =$ Acquire $\wedge$ $\forall i, i \xrightarrow{\mathtt{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R})$	$m_1 =$ Release$\wedge$ $m_2 \sqsupseteq$ Release$\vee$ $(m_1 =$ Acquire $\wedge$ $\forall i, i \xrightarrow{\mathtt{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R})$	$(m_1 =$ Release $\wedge$ $m_2 =$ Acquire$) \vee (m_1 =$ Acquire $\wedge$ $\forall i, i \xrightarrow{\mathtt{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R}) \vee (m_2 =$ Release $\wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\mathtt{po}} i \Rightarrow i \notin H.\mathtt{W})$

Figure 6 Allowed Deordering Pairs in JAM_{21}.

Proof. Assume towards contradiction that H_{src} is not JAM_{21}-consistent. Then there are two cases: either there is a $(\mathtt{po}\,|\,\mathtt{rf})^+$ cycle or a co cycle in H_{src}. Whether or not a and b are included in this cycle, adding a po edge between a and b cannot eliminate this cycle (although it might introduces new cycles). Therefore, H_{tgt} is also not JAM_{21}-consistent, contradicting to our assumption. ◀

5.3 Reordering

The operation of *reordering* can be seen as composing *deordering* with *sequentialisation*. Since we know that sequentialisation is sound in JAM_{21}, we only need to show that deordering is sound in order to show reordering is sound in JAM_{21}.

Deordering

Deordering is a transformation that turns a pair of accesses related by a po relation into a pair of concurrent accesses. In effect, it removes an po edge in the execution graph.

First, we adopt the same definition of adjacent events from [6]:

▶ **Definition 14** (Adjacent Events). *Two events a and b are **adjacent** in a partial order* R *if for all c, we have:*

- $c \xrightarrow{\mathtt{R}} a \Rightarrow c \xrightarrow{\mathtt{R}} b$
- $b \xrightarrow{\mathtt{R}} c \Rightarrow a \xrightarrow{\mathtt{R}} c$

For Java, the table of allowed reordering two adjacent events (with each row as the first event and column as the second event) is shown in Fig. 6 (some of the cases are different from C11 [6] and we have marked them in red). Intuitively, the sound deorderable pairs are ordered by the po edges that does not impose any synchronization in the program. Therefore, deordering (removing the po edge) does not introduce new program behavior.

Name	C/C++11	Java
Read-read Merging	$\mathtt{R}^m; \mathtt{R}^m \leadsto \mathtt{R}^m$	$\mathtt{R}^{m \sqsubseteq \mathsf{Acq}}; \mathtt{R}^{m \sqsubseteq \mathsf{Acq}} \leadsto \mathtt{R}^m$
Write-write Merging	$\mathtt{W}^m; \mathtt{W}^m \leadsto \mathtt{W}^m$	$\mathtt{W}^{m \sqsubseteq \mathsf{Rel}}; \mathtt{W}^{m \sqsubseteq \mathsf{Rel}} \leadsto \mathtt{W}^m$
Write/RMW-read Merging	$\mathtt{W}^m; \mathtt{R}^{\mathsf{acq}} \leadsto \mathtt{W}^m$	$\mathtt{W}^m; \mathtt{R}^{m \sqsubseteq \mathsf{Opq}} \leadsto \mathtt{W}^m$
	$\mathtt{W}^{\mathsf{sc}}; \mathtt{R}^{\mathsf{sc}} \leadsto \mathtt{W}^{\mathsf{sc}}$	✗
	$\mathtt{RMW}^m; \mathtt{R}^{m_r \sqsubseteq m} \leadsto \mathtt{RMW}^m$	$\mathtt{RMW}^m; \mathtt{R}^{m \sqsubseteq \mathsf{Opq}} \leadsto \mathtt{RMW}^m$
Write-RMW Merging	$\mathtt{W}^{m_w \sqsubseteq m}; \mathtt{RMW}^m \leadsto \mathtt{W}^{m_w}$	$\mathtt{W}^{m_w \sqsubseteq \mathsf{Rel}}; \mathtt{RMW}^{m \sqsubset \mathsf{Vol}} \leadsto \mathtt{W}^{m_w}$
RMW-RMW Merging	$\mathtt{RMW}^m; \mathtt{RMW}^m \leadsto \mathtt{RMW}^m$	$\mathtt{RMW}^{m \sqsubset \mathsf{Vol}}; \mathtt{RMW}^{m \sqsubset \mathsf{Vol}} \leadsto \mathtt{RMW}^m$
Fence-fence Merging	$\mathtt{F}^m; \mathtt{F}^m \leadsto \mathtt{F}^m$	$\mathtt{F}^m; \mathtt{F}^m \leadsto \mathtt{F}^m$

Figure 7 Mergable Pairs in C/C++11 [6] and Java.

To prove that JAM_{21} supports the reordering shown in this table, we need to prove each cell shown in the table is valid for JAM_{21}.

▶ **Theorem 15** (Deordering). *Let P_{src} be a Java program and P_{tgt} be a Java program obtained by performing a deordering operation on a pair of accesses a and b according to Fig. 6. Let H_{tgt} be an execution of P_{tgt}. Then there exists an execution H_{src} of P_{src} such that*

- $H_{src}.\mathsf{po} = H_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\}$ *where a and b are* po*-adjacent*
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$
- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E}$
- $H_{src}.\mathit{to} = H_{tgt}.\mathit{to}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$
- $\forall i \in H_{src}.\mathtt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$

and if H_{tgt} is JAM_{21}-consistent, then H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

Reordering, as mentioned previously, can be decomposed into two steps: deordering and sequentialisation. Since we have already shown the soundness of the two transformations, the soundness of reordering follows naturally.

▶ **Corollary 16** (Reordering). *JAM_{21} supports the reordering transformation for pairs of adjacent accesses shown in Fig. 6.*

5.4 Merging

Merging transforms two adjacent accesses into one single equivalent access to reduce the number of memory accesses in the program. We have grouped all types of merging transformations appeared in C/C++11 [6] here in one section. A summarized result of mergable pairs comparing with C/C++11 can be found in Fig. 7. The results are mostly similar except for Volatile. Many merging transformation are invalid for Volatile because they remove the cross-thread synchronization of Volatile.

5.4.1 Read-Read Merging

Read-read merging is sometimes done when the compiler is optimizing redundant loads in the same thread. When we are encountering two consecutive reads to the same location, the first read is unchanged but the second read becomes a local read without accessing the memory.

Let a' and b be two adjacent read accesses reading from the same write access a. $a \xrightarrow{\texttt{rf}} a'$ and $a \xrightarrow{\texttt{rf}} b$, and $a' \xrightarrow{\texttt{po}} b$. Assuming $AccessMode(a') = AccessMode(b)$, then

- $\forall i, a' \xrightarrow{\texttt{po}} i \Rightarrow b \xrightarrow{\texttt{po}} i$
- $\forall i, a' \xrightarrow{\texttt{ra}} i \Rightarrow b \xrightarrow{\texttt{ra}} i$
- $\forall i, a' \xrightarrow{\texttt{push}} i \Rightarrow b \xrightarrow{\texttt{push}} i$
- $\forall j, j \xrightarrow{\texttt{po}} b \Rightarrow j \xrightarrow{\texttt{po}} a'$

For executions, this corresponds to the following transformation in the execution graph: since the value of $r1$ and $r2$ are guaranteed to have the same value in P_{tgt}, we know that this corresponds to the execution of P_{src} where the two read accesses read from the same write access. Then we want to show that, if H_{tgt} is JAM_{21}-consistent, H_{src} is also JAM_{21}-consistent.

▶ **Theorem 17** (Read-Read Merging). *Let H_{tgt} be an JAM_{21}-consistent execution. Let $a \in H_{tgt}.R \backslash RMW$ and let $a' \in H_{tgt}.\texttt{E}$ such that $a \xrightarrow{\texttt{rf}} a'$. Let $b \notin H_{tgt}.\texttt{E}$. There exists a H_{src} such that:*

- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\texttt{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\texttt{po}} j\}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf} \cup \{\langle a', b\rangle\}$
- $H_{src}.\texttt{E} = H_{tgt}.\texttt{E} \cup \{b\}$
- $H_{src}.to = H_{tgt}.to \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{to} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{to} j\}$
- $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$
- $\forall i \in H_{tgt}.\texttt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $b \in H_{src}.R$
- $H_{src}.AccessMode(b) = H_{src}.AccessMode(a) \sqsubseteq$ *Acquire*

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

Note that JAM_{21} does not allow read-read merging if the two read accesses are both Volatile mode reads. We provide an example of this in the full version of this paper.

5.4.2 Write-Write Merging

The write-write merge transformation refers to the program transformation that merges two consecutive write operations into one by removing the former one. JAM_{21} support write-write merge when the access modes of the two writes are the same and they are not Volatile mode accesses.

Let a and b be the two adjacent writes such that $a \xrightarrow{\texttt{po}} b$. We once again have the properties:

- $\forall i, i \xrightarrow{\texttt{po}} a \Rightarrow i \xrightarrow{\texttt{po}} b$
- $\forall j, b \xrightarrow{\texttt{po}} j \Rightarrow a \xrightarrow{\texttt{po}} j$
- $\forall i, i \xrightarrow{\texttt{ra}} a \Rightarrow i \xrightarrow{\texttt{ra}} b$

We have the following theorem.

▶ **Theorem 18** (Write-Write Merging). *Let H_{tgt} be an JAM_{21}-consistent execution. Let $b \in H_{tgt}.W \backslash RMW$ and let $a \notin H_{tgt}.\texttt{E}$ and $loc(a) = loc(b) \wedge \forall i \in H_{tgt}.W, loc(i) = loc(b) \Rightarrow val(a) \neq val(i)$. There exists a H_{src} such that:*

- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\texttt{po}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\texttt{po}} j\}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf}$
- $H_{src}.\texttt{E} = H_{tgt}.\texttt{E} \cup \{a\}$
- $H_{src}.\textit{to} = H_{tgt}.\textit{to} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{to} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{to} j\}$
- $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$
- $\forall i \in H_{tgt}.\texttt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $a \in H_{src}.W$
- $H_{src}.AccessMode(a) = H_{src}.AccessMode(b) \sqsubseteq \textit{Release}$

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

Note that write-write merging is not valid for Volatile mode writes. We provide an example of this in the full version of this paper.

5.4.3 Write/RMW-read Merging

The Write/RMW-read merging refers to the program transformation that merges a write/ RMW and a read into a single write/RMW and a local access.

Similarly, the transformation with an RMW operation and a read operation optimizes the latter read operation to read locally and in effect removes a memory load operation in the execution graph.

JAM_{21} only support this transformation when the read operation is (or is weaker than) Opaque mode which is different from RC11 [6]'s result for C/C++11. We provide a counter-example in the full version of this paper to show that write/RMW-read merging is invalid when the read is (or is stronger than) Acquire mode.

▶ **Theorem 19** (Write/RMW-Read Merging)**.** *Let H_{tgt} be a JAM_{21}-consistent execution. Let $a \in H_{tgt}.\texttt{W}$ and $b \notin H_{tgt}.\texttt{E}$. There exists a H_{src} such that:*

- $H_{src}.\texttt{E} = H_{tgt}.\texttt{E} \cup \{b\}$
- $b \in H_{src}.\texttt{R}$
- $H_{src}.loc(b) = H_{src}.loc(a)$
- $H_{src}.val(b) = H_{src}.val(a)$
- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\texttt{po}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\texttt{po}} j\}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf} \cup \{\langle a,b\rangle\}$
- $H_{src}.\textit{to} = H_{tgt}.\textit{to} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{to} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{to} j\}$
- $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$
- $\forall i \in H_{tgt}.\texttt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $H_{src}.AccessMode(b) \sqsubseteq \textit{Opaque}$

Please see the full version of this paper for the proof.

5.4.4 Write-RMW Merging

The write-RMW merging refers to the program transformation that merges a write and a consecutive RMW operation into a write with the value of the RMW. For example, if we have the following pattern in a program:

```
x = 1;
x.getAndSet(1,2);
```

It can be tranformed to:

```
x = 2;
```

Similar to write-write merging, JAM_{21} supports write-RMW merging when the access mode of the write is {Opaque, Release} and the access mode of the RMW is {Acquire, Release}.

▶ **Theorem 20** (Write-RMW Merging). *Let H_{tgt} be a JAM_{21}-consistent execution. Let $b \in H_{tgt}.\mathtt{W} \setminus H_{tgt}.\mathtt{RMW}$, $a \notin H_{tgt}.\mathtt{E}$ and $v \in \mathtt{Val}$. There exists a H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E} \cup \{a\}$
- $\forall i \in H_{tgt}.\mathtt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $H_{src}.AccessMode(a) \in \{\mathit{Opaque}, \mathit{Release}\}$
- $H_{src}.AccessMode(b) \in \{\mathit{Acquire}, \mathit{Release}\}$
- $H_{src}.loc(b) = H_{src}.loc(a)$
- $b \in H_{src}.\mathtt{RMW}$
- $H_{src}.val(b) = (H_{src}.val(a), v)$
- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf} \cup \{\langle a, b\rangle\}$
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

5.4.5 RMW-RMW Merging

The RMW-RMW merging transformation refers to the program transformation that merges two consecutive RMW operations into one such that it has the first RMW's (expected) read value and the second RMW's write value. For example, if we have the following pattern in a program:

```
x.getandSet(1,2);
x.getandSet(2,3);
```

then it might be transformed into:

```
x.getAndSet(1,3);
```

The RMW-RMW merging transformation is essentially the same as write-write merging and read-read merging described previously. Therefore, the set of constraints on valid access modes for merging is the intersection of the two. That is, two RMWs are mergeable if they are both Acquire mode or Release mode. For the counter-examples showing this transformation is invalid for Volatile accesses, please see the examples for write-write and read-read merging.

▶ **Theorem 21** (RMW-RMW Merging). *Let H_{tgt} be a JAM_{21}-consistent execution. Let x be a memory location and $a \in H_{tgt}.\mathtt{E}$ with $H_{tgt}.val(a) = (v_r, v_w)$, $H_{tgt}.loc(a) = x$, and $H_{tgt}.AccessMode(a) \in \{\mathit{Release}, \mathit{Acquire}\}$. Let $b \notin H_{tgt}.\mathtt{E}$, there exists a H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E} \cup \{b\}$
- $\forall i \in H_{tgt}.\mathtt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $H_{src}.val(a) = (v_r, v)$
- $H_{src}.val(b) = (v, v_w)$
- $H_{src}.loc(b) = x$
- $H_{src}.AccessMode(b) = H_{src}.AccessMode(a) \in \{\mathit{Release}, \mathit{Acquire}\}$
- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathtt{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathtt{po}} j\}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf} \cup \{\langle a, b\rangle\}$
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathtt{to}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathtt{to}} j\}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

5.4.6 Fence-fence Merging

The Fence-fence merging refers to the program transformation that merges two consecutive fences of the same access mode into one. For example, if we have:

```
VarHandle.fullFence();
VarHandle.fullFence();
```

then it can be optimized to:

```
VarHandle.fullFence();
```

Since JAM_{21} is fence-based such that each fence is converted into an edge between memory accesses, this is trivially supported since the execution graph before and after the transformation is exactly the same.

5.5 Register Promotion for Non-shared Variable

Register Promotion promotes memory accesses of a non-shared memory location to local registers. It has the effect of removing memory accesses for thread-local variables. JAM_{21} only supports register promotion for variables without any Volatile accesses in the program. For non-Volatile accesses, since the variable is not shared across threads, it is safe to remove them without worrying about removing synchronization from the program. In contrast, Volatile accesses impose cross-thread synchronizations with Volatile accesses for other variables, so removing such accesses can potentially remove important synchronization in the program and introduce new behaviors that were previously forbidden by the memory model. We provide a counter-example in this section showing that we cannot promote Volatile accesses to local register accesses even if the location is only accessed by one thread.

Suppose all accesses to a memory location are in the same thread, the transformation can be seen as two steps:

1. Weakening the accesses to Plain mode accesses
2. Removing the Plain mode accesses

▶ **Theorem 22** (Weakening for non-shared variable). *Let H_{tgt} be a JAM_{21}-consistent execution such that, for all accesses i and j in $H_{tgt}.\mathtt{E}$, $loc(i) = loc(j) = x \Rightarrow Tid(i) = Tid(j)$ for some memory location x. In addition, $\forall i \in H_{tgt}.\mathtt{E}, loc(i) = x \Rightarrow AccessMode(i) = \textsf{Plain}$. There exists an execution H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E}$
- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$
- $\forall i \in H_{src}.\mathtt{E}, loc(i) = x \Rightarrow AccessMode(i) \in \{\textsf{Release}, \textsf{Acquire}\}$

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

▶ **Theorem 23** (Removing Plain accesses for non-shared variable). *Let H_{tgt} be a JAM_{21}-consistent execution. Let x be a memory location and for all $i \in H_{tgt}.\mathtt{E}$ such that $loc(i) = x$, $Tid(i) = t$ for some t. Let $a \notin H_{tgt}.\mathtt{E}$. There is a H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E} \cup \{a\}$
- $H_{src}.loc(a) = x$
- $H_{src}.AccessMode(a) = \textsf{Plain}$
- $H_{src}.\mathtt{po} \supset H_{tgt}.\mathtt{po}$
- *for all* $i \in H_{src}.\mathtt{E}$ *such that* $H_{src}.loc(i) = x$, $i \xrightarrow{\mathtt{po}} a$ *or* $a \xrightarrow{\mathtt{po}} i$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$ *if* $a \in H_{src}.W \backslash RMW$, *otherwise,* $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf} \cup \{\langle i, a\rangle\}$ *such that* $(i \in H_{src}.W) \wedge (loc(i) = x) \wedge (i \xrightarrow{\mathtt{po}} a) \wedge (\forall j \in H_{src}.\mathtt{E}, (loc(j) = x) \wedge (j \xrightarrow{\mathtt{po}} a) \Rightarrow (j \xrightarrow{\mathtt{po}} i))$.
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$

and H_{src} *is* JAM_{21}*-consistent.*

Please see the full version of this paper for the proof.

Counter Example

We now show a counter example for invalid register promotion on locations with Volatile accesses. Consider the following program:

```
Thread0 {
  int r1 = X.getOpaque(); // 1
  int r2 = X.getOpaque(); // 2
}

Thread1 {
  int r3 = Y.getOpaque(); // 1
  int r4 = Y.getOpaque(); // 2
}

Thread2 {
  X.setOpaque(2);
  Z.setVolatile(1);
  Y.setVolatile(1);
}

Thread3 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

An execution with the annotated values in this program is not allowed by JAM_{21}. The execution graph before the transformation is shown in Fig. 8. First note that the Volatile access on z also has Release semantics due to the monotonicity of access modes, which yields the ra edge in Thread 2. The total order among push edges gives use two cases:

1. `Wz = 1` $\xrightarrow{\mathtt{vvo}}$ `Wx = 1`. Since `Wx = 2` $\xrightarrow{\mathtt{ra}}$ `Wz = 1` and $\mathtt{ra} \subseteq \mathtt{vvo}$ and $\mathtt{vvo}^+ \subseteq \mathtt{vo}$, we have `Wx = 2` $\xrightarrow{\mathtt{vo}}$ `Wx = 1`, which contradict with the co edge established by the observation from Thread 0.
2. `Wy = 2` $\xrightarrow{\mathtt{vvo}}$ `Wy = 1`. This contradict with the co edge established by the observation from Thread 1.

In both cases there is a contradiction (a co cycle). Therefore, this execution is forbidden by JAM_{21}.

In this example, the memory location z is only accessed by Thread 2. It maybe tempting to promote z to a local register on Thread 2 to reduce the number of memory instructions, which yields the following program:

```
Thread0 {
  int r1 = X.getOpaque();
  int r2 = X.getOpaque();
}

Thread1 {
  int r3 = Y.getOpaque();
  int r4 = Y.getOpaque();
}

Thread2 {
  X.setOpaque(2);
  int z = 1
  Y.setVolatile(1);
}

Thread3 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

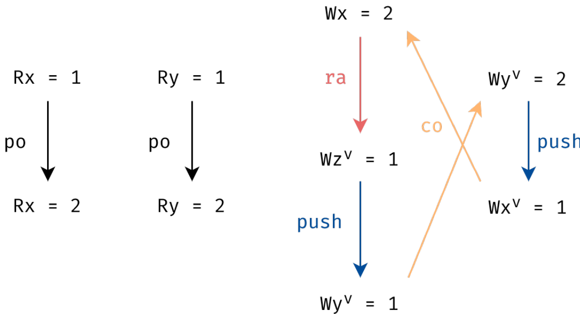


Figure 8 Before Register Promotion on Volatile access (Forbidden).

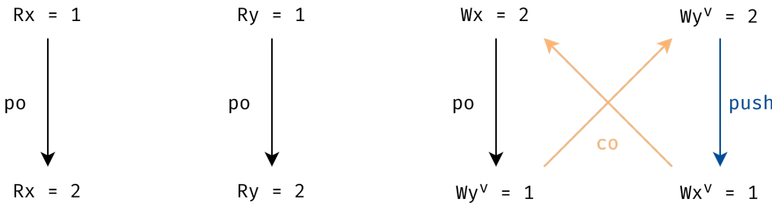


Figure 9 After Register Promotion on Volatile access (Allowed).

The execution graph after the transformation is shown in Fig. 9.

The annotated program behavior becomes allowed by JAM_{21} after the transformation. As the execution graph shows, since Volatile accesses also have cross-thread synchronization effect, we cannot simply weaken it to a Plain access without introducing new program behaviors.

5.6 Why are many transformations invalid for Volatile?

As we have shown, many local transformations are invalid for Volatile accesses under JAM_{21}. This is not a surprise and is intended to provide programmers a more intuitive semantics for Volatile accesses.

First, as we have confirmed with the author of [9], Java's Access Modes intend equivalent semantics for Volatile mode and `fullFence()`. In this way, the programmers can easily understand the semantics of both once they understand `fullFence()`. To accurately capture this intention, JAM_{21} used a fence-based approach with push order to model Volatile mode. As we described in Section 3, `fullFence()` in Java has cross-thread synchronization effects. As a result, any local program transformation that removes a Volatile access from the execution graph may also remove its cross-thread synchronization, and might introduce new program behavior after the transformation. Therefore, those transformations on Volatile accesses are mostly not allowed by JAM_{21}. On the other hand, the `sc` fence in C/C++11 [6] has slightly stronger synchronization effect than `sc` accesses so that they can be used to restore sequential consistency when inserted between every pair of accesses. Some of the transformations are allowed to apply to `sc` accesses but not to the fence version of the program.

In addition, restricting the set of possible transformations that is allowed to apply to Volatile variables can keep the coding process simple for programmers. From the programmers' perspective, one of the biggest challenges of developing and debugging concurrent programs

comes from the compiler transformations that introduces surprising program behaviors that are not observable under sequential consistency. Therefore, restricting the set of possible transformations on Volatile accesses can restrict the set of surprising program behaviors that can happen when using Volatile mode, making the development process simpler. From this perspective, JAM_{21} provides more synchronization guarantees for Volatile mode than C/C++11 for `sc` mode atomic accesses.

Lastly, as we have confirmed with the author of [9], the current implementation of OpenJDK JVM does not apply those transformations on Volatile accesses.

6 Performance Implications

At the time of writing, the compiler bug [17] has been reported but still not resolved. The main argument against fixing the bug by inserting the missing fence instruction is that it may slow down the performance significantly. In this section, we argue that this is not the case.

The reason we only translated our `volatile-non-sc` example to Power instructions is that we only expect changes in the implementation of compilers targeting Power architectures. There is no need to change the Java compilers for x86 [15] and ARMv8 [13] all thanks to a property called *write atomicity*. Write atomicity, or *multicopy atomicity*, ensures that, when a write issued by a thread becomes observable by any other thread, it is observable by all other threads in the system. The issue that we demonstrate in this paper is caused by a write operation becoming visible to some threads before some other threads. Therefore, this violation of sequential consistency may only be observed when compiling to non-multicopy atomic architectures. If the underlying architecture ensures multicopy atomicity, then we can be sure that all writes are committed in a broadcast style and Release-Acquire semantics is sufficient. Since x86 [15] and ARMv8 [13] are multicopy atomic, we do not expect the incorrect program behavior to appear on those architectures. Therefore, no change is needed in compilers targeting multicopy-atomic architectures. In fact, we give a correctness proof for x86 in the full version of this paper to concretely show that the current compilation scheme to x86 is correct with respect to the x86-TSO memory model. Furthermore, the fence instruction that compilers use to compile to ARMv7 is the `DMB SY` instruction [8], which captures the same effects of a `fullFence()`. The only change that needs to be made is when compiling to Power instructions. This change might slow down some programs. However, relative to all other major factors that affect the performance of Java programs, we expect the impact by this change in compilers to be small.

Furthermore, symmetric to "leading fence" scheme, the "trailing fence" scheme is also valid. A correct compiler may choose to either of the schemes. Usually one may wish to choose the "trailing fence" scheme for better performance. In this case, comparing to the original compilation scheme, the fix only changes the compilation scheme for each Volatile read:

1. Remove the `hwsync` in front of the `lwz` instruction
2. Change the `lwsync` following the `lwz` instruction to `hwsync`

It is easy to see that this fix only requires, in effect, moving the `hwsync` instructions that were originally inserted before the `lwz` instruction, but does not add more. In addition, it removes the `lwsync` instructions. Therefore, we do not expect this change to the compilation scheme to have much performance impact as argued in the discussions in the bug report [17].

On the other hand, the impact of this change for compiler optimizations is unclear. That is, whether this revised compilation scheme disables some of the compiler optimizations is still a question. However, since C/C++11 compilers has long adopted this compilation

scheme and performance has always been the first priority in their implementations, the possibility of disabling optimisations is unlikely. We leave a detailed empirical study for future work.

7 Related Work

7.1 Sequential Consistency Issue in C/C++11

A similar but different issue in C/C++11 memory model for atomic operations with sequentially consistent memory order was pointed out by Manerkar, et al. [11] and Lahav, et al. [6]. In particular, when using the "trailing fence" convention for compiling to Power and ARMv7 on GCC, the intended sequentially consistent semantics for certain atomic accesses can be lost due to the different placement of fences in the programs. In other words, the previous C/C++11 memory model was not able to support the two existing compilation schemes on GCC. On the other hand, JAM_{19} did not have the same problem. Since JAM_{19} defined the semantics of Volatile mode in terms of `push` orders, which emulates the effect of a full fence, it already supports and aligned with the existing compilation scheme found on OpenJDK JVMs.

The problem, however, was that the existing compilation scheme does not give sufficient synchronization to some programs with all accesses marked as Volatile. Since JAM_{19} models the problematic compilation scheme, it is necessary to repair the problem for both the compiler and the formal model.

7.2 Using Volatile to Restore Sequential Consistency in Java

Due to the complexity of the original Java Memory Model (JMM) [12], a class of bugs caused by missing "`volatile`" annotations on certain shared variables, called *missing-annotation bugs*, is found across real-world Java applications [10]. Aiming to improve the safety guarantees of the Java language, volatile-by-default JVM was proposed and developed by [10] to advocate the idea that variables should have `volatile` semantics by default and relaxed semantics by choice. Following their idea, the correctness of volatile (or Volatile mode, as they are equivalent) semantics become especially important. After all, if we cannot restore sequential consistency by annotating every variable as `volatile` (or use Volatile mode for every access), then volatile-by-default JVM would not be able to ensure intuitive program behaviors either. As of today, we are not aware of any `volatile`-by-default JVM for versions of Java after JDK9. Thus, we suggest that researchers carefully ensure the correctness of the `volatile` (or Volatile mode) implementations when implementing such JVM for Java versions after JDK9.

7.3 Memory Fairness and Compiler Transformations

Recently a declarative definition of *memory fairness* was proposed for axiomatic relaxed memory models [5]. As an improvement to the existing definition of *thread fairness*, the declarative memory fairness property can be easily integrated into axiomatic models with the No-Thin-Air restriction and can be used to prove the termination of concurrent algorithms. We noticed that the original JAM model [3] was published before this definition was proposed and therefore did not make any assertions regarding memory fairness. We leave it as our future work to verify whether memory fairness preserves the correctness of the compiler transformations and the compilation schemes.

8 Conclusion

In this paper, we have demonstrated that Java can use a compilation scheme that is similar to C/C++11. On the other hand, one should not simply compile Java's Access Modes the same way as C/C++11 compiles atomic memory orders since the formal memory models supports different compiler optimizations. In the future, we hope the bug can be resolved soon and the examples in this paper can be added to the Java Concurrency Stress Tests *jcstress* [16] tool suite to aid in maintaining the correctness of the OpenJDK HotSpot implementations.

References

1. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), July 2014. doi:10.1145/2627752.
2. ARM ARM. Architecture reference manual-armv8, for armv8-a architecture profile. *ARM Limited, Dec*, 2017.
3. John Bender and Jens Palsberg. A formalization of java's concurrent access modes. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360568.
4. Peter Sewell Jaroslav Sevcik. C/C++11 mappings to processors. Technical report, University of Cambridge, October 2016. URL: https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.
5. Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. Making weak memory models fair. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485475.
6. Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062352.
7. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979. doi:10.1109/TC.1979.1675439.
8. Doug Lea. The jsr-133 cookbook for compiler writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html, 2011. Last modified: Tue Mar 22 07:11:36 2011.
9. Doug Lea. Using jdk 9 memory order modes. http://gee.cs.oswego.edu/dl/html/j9mm.html, 2018. Last Updated: Fri Nov 16 08:46:48 2018.
10. Lun Liu, Todd Millstein, and Madanlal Musuvathi. A volatile-by-default jvm for server applications. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133873.
11. Yatin A Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the c/c++ to power and armv7 trailing-sync compiler mappings. *arXiv preprint arXiv:1611.01507*, 2016.
12. Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, January 2005. doi:10.1145/1047659.1040336.
13. Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158107.
14. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993520.
15. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010. doi:10.1145/1785414.1785443.

16 Aleksey Shipilev. jcstress - the java concurrency stress tests. https://wiki.openjdk.java.net/display/CodeTools/jcstress, 2017. Last Updated: Wed Dec 05 13:55 2018.

17 Aleksey Shipilev. [JDK-8262877] PPC sequential consistency problem: volatile stores are too weak. Technical report, OpenJDK Bug System, March 2021. URL: https://bugs.openjdk.java.net/browse/JDK-8262877.

Functional Programming with Datalog

André Pacak
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

Datalog is a carefully restricted logic programming language. What makes Datalog attractive is its declarative fixpoint semantics: Datalog queries consist of simple Horn clauses, yet Datalog solvers efficiently compute all derivable tuples even for recursive queries. However, as we argue in this paper, Datalog is ill-suited as a programming language and Datalog programs are hard to write and maintain. We propose a "new" frontend for Datalog: functional programming with sets called *functional IncA*. While programmers write recursive functions over algebraic data types and sets, we transparently translate all code to Datalog relations. However, we retain Datalog's strengths: Functions that generate sets can encode arbitrary relations and mutually recursive functions have fixpoint semantics. We also ensure that the generated Datalog program terminates whenever the original functional program terminates, so that we can apply off-the-shelve bottom-up Datalog solvers. We demonstrate the versatility and ease of use of functional IncA by implementing a type checker, a program transformation, an interpreter of the untyped lambda calculus, two data-flow analyses, and clone detection of Java bytecode.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Datalog, functional programming, demand transformation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.7

1 Introduction

Datalog is a carefully restricted logic programming language that has seen a surge in popularity in recent years. Originally, Datalog was conceived as a database query language that operates on finite sets only [15], so that all queries are guaranteed to terminate. Nowadays, Datalog is being used in a wide array of applications [12], from program analysis [10, 13, 23] to network monitoring [1] and distributed computing [2, 3]. What makes Datalog so popular is that (i) there are highly efficient and scalable implementations available and (ii) Datalog programs are considered declarative. We argue that the latter is partly a misconception: Datalog's semantics is declarative, but Datalog's frontend is not.

Datalog is often primed as being declarative. This can be surprising given that a Datalog program consists of simple Horn clauses (a_0 :- $a_1, \ldots, a_n$), where a_0 holds if a_1 through a_n hold. In Datalog, a_0 is called the *head* of the rule and $a_1, \ldots, a_n$ form the *body* of the rule. Both head and body consist of *atoms* a, which are of the form $R(t_1, \ldots, t_n)$ for some relation R and terms t. A Datalog solver computes the least fixpoint of the Horn clauses such that the relations R contain all derivable ground tuples, called *facts* in Datalog. In the initial fixpoint iteration, the semantics collects all rule heads a_0 that have no precondition. In subsequent fixpoint iterations, the semantics collects all facts that can be derived by applying rules to previously derived facts. When terms range over finite sets, this fixpoint iteration terminates in finitely many steps. We concur that Datalog has a declarative semantics, because programmers do not need to think about *how* the derivable facts are computed.

The problem of Datalog is its frontend: It is ill-suited as a programming language and not declarative. Consider we want to construct control-flow graphs as a basis for program analysis. Figure 1 shows a functional program and a Datalog program that construct the

© André Pacak and Sebastian Erdweg;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 7; pp. 7:1–7:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
// Functional programming
def flow(stm: Stm): Set[(Stm, Stm)] = stm match {
  case Assign(x, a) => {}
  case Sequence(s1, s2) => flow(s1) ++ flow(s2) ++ {(l1, init(s2)) | l1 in final(s1)}
  case If(c, s1, s2) => c match {
    case True() => flow(s1) ++ {(stm, init(s1))}
    case False() => flow(s2) ++ {(stm, init(s2))}
    case _ => flow(s1) ++ flow(s2) ++ {(stm, init(s1)), (stm, init(s2))} }
  case While(c, s) => flow(s) ++ {(stm, init(s))} ++ {(l,stm) | l in final(s)} }
```

```
// Datalog
flow(Stm, From, To) :- sequence(Stm, Stm1, _), flow(Stm1, From, To).
flow(Stm, From, To) :- sequence(Stm, _, Stm2), flow(Stm2, From, To).
flow(Stm, From, To) :- sequence(Stm, Stm1, Stm2), final(Stm1, From), init(Stm2, To).
flow(Stm, From, To) :- if(Stm, C, Stm1, _), true(C), flow(Stm1, From, To).
flow(Stm, Stm, To) :- if(Stm, C, Stm1, _), true(C), init(Stm1, To).
flow(Stm, From, To) :- if(Stm, C, _, Stm2), false(C), flow(Stm2, From, To).
flow(Stm, Stm, To) :- if(Stm, C, _, Stm2), false(C), init(Stm2, To).
flow(Stm, From, To) :- if(Stm, C, Stm1, _), not true(C), not false(C), flow(Stm1,From,To).
flow(Stm, Stm, To) :- if(Stm, C, Stm1, _), not true(C), not false(C), init(Stm1,To).
flow(Stm, From, To) :- if(Stm, C,_,Stm2), not true(C), not false(C), flow(Stm2,From,To).
flow(Stm, Stm, To) :- if(Stm, C,_,Stm2), not true(C), not false(C), init(Stm2,To).
flow(Stm, From, To) :- while(Stm, _, Stm1), flow(Stm1, From, To).
flow(Stm, Stm, To) :- while(Stm, _, Stm1), init(Stm1, To).
flow(Stm, From, Stm) :- while(Stm, _, Stm1), final(Stm1, From).
```

Figure 1 Constructing control-flow graphs using functional programming and Datalog.

control-flow graphs for the While language. The functional program uses pattern matching and set-comprehensions to compute sets of edges similar to [16], whereas the Datalog program provides rules to constrain the logic variables `From` and `To`. The most prominent problem with Datalog in this example is the lack of structured programming and the duplication of atoms, especially for *if*-statements: we must query relation `if` 8 times, relation `true` 6 times, and relation `false` 6 times. Such Datalog code is hard to write and maintain. Another problem with programming Datalog is that rules must be *range-restricted*: Each variable in the head of a rule must be bound in the body of the rule. This restriction ensures relations can be computed using Datalog's least fixpoint semantics. For example, the increment relation `inc(X, Y) :- Y=X+1` would be correctly rejected by Datalog solvers such as Soufflé [20], because `X` is not bound in the rule's body. Datalog programmers need to work around this restriction.

So why would programmers want to use Datalog anyways instead of functional programming? Because of Datalog's declarative fixpoint semantics, which makes it easy to process cyclic data structures such as our control-flow graphs from above. For example, we can compute the transitive control-flow reachability with two simple rules:

```
flowTrans(Prog, From, To) :- flow(Prog, From, To).
flowTrans(Prog, From, To) :- flow(Prog, From, Inter), flowTrans(Prog, Inter, To).
```

What is remarkable is that we do not have to implement a termination condition and or detect when the relations are stable; Datalog takes care of that. This is why Datalog is a popular implementation language for data-flow analyses that propagate information along the control-flow graph until reaching a fixpoint [10], despite the shortcomings of its frontend.

In this paper, we design a functional programming language with fixpoint semantics and propose it as a "new" Datalog frontend: *functional IncA*. In particular, we show how functional programs with first-order functions and recursive algebraic data types can be

```
def entry_var(stm: Stm, prog: Stm, x: String): Val =
  fold(BotVal(), joinVal, {exit_var(pred, prog, x) | (pred,stm) in flow(prog)})
def exit_var(stm: Stm, prog: Stm, x: String): Val = stm match {
  case Assign(y, exp) =>
    if (x == y)  aeval(exp, stm, prog)
    else         entry_var(stm, prog, x)
  case ... }
def aeval(exp: Exp, node: Stm, prog: Stm): Val = exp match {
  case Var(x) => entry_var(node, prog, x)
  case ... }
```

Figure 2 A data-flow analysis using functional programming with fixpoint semantics.

faithfully translated to negation-free Datalog. A key idea of our approach is to systematically track the *demand* on functions: Which inputs must a function be run on to obtain the computation's final result. Since terminating functional programs only consider finitely many inputs, we can track these inputs in Datalog relations. For programs without algebraic data types, we can adopt a standard demand transformation [25]. However, for algebraic data types we need to carefully instrument the demand transformation to encode constructors and selectors through finite relations. Our translation preserves the semantics of the functional program and, in particular, the resulting Datalog program terminates whenever the functional program does. The translation targets Datalog with base types and operations on such types such as integers, float, strings, booleans as well as algebraic data types. The most common Datalog dialects such as Soufflé, IncA, Flix and Formulog all support these types. Whenever we reference Datalog we mean Datalog with the extensions listed above.

Functional IncA replaces Datalog's logic programming frontend, but we retain Datalog's key advantages: relations with fixpoint semantics. Specifically, we extend functional IncA with set types and set operations (comprehensions, union, and folds) such that programmers can describe and aggregate over relations. For example, Figure 2 shows a data-flow analysis implemented in functional IncA. The analysis queries the control-flow graph `flow` and propagates information about the value of variables (abstracted as intervals) along the control-flow graph. Note that `entry_var`, `exit_var`, and `aeval` are mutually recursive and that there is no termination condition (the program diverges under standard functional semantics). Despite using functional programming as a frontend, all code compiles to Datalog rules, which is a key advantage of our approach for two reasons. First, programmers can rely on the declarative Datalog semantics to find the least fixpoint. Second, we can use any existing Datalog solver to run the program, whereas prior Datalog dialects usually require a custom Datalog solver.

We have implemented functional IncA as part of the incremental Datalog framework IncA [23]. We translate functional IncA into a Datalog IR and provide two backends: one targeting IncA directly, the other targeting Soufflé [20]. While the IncA backend provides incremental re-evaluation after input changes, the Soufflé backend provides better non-incremental performance. The choice of the backend is transparent for the user of the frontend, except that Soufflé does not support user-defined aggregations. We have implemented three case studies using functional IncA. First, we implemented a type checker for the simply-typed lambda calculus, a type-erasure transformation for the same, and an interpreter for the untyped lambda calculus. While the encoding of type checkers in Datalog has recently been explored [17], we are the first to support program transformations and interpreters for Turing-complete languages in Datalog without relying on an embedded functional programming language. Second, we implemented textbook reaching definitions and interval analyses. Both analyses are flow-sensitive and compute a fixpoint over the

control-flow graph. Last, we implement clone detection of Java bytecode which is represented as Soufflé facts. We generate abstract syntax trees by querying Soufflé relations. We then use the abstract syntax trees to determine if two methods are alpha-equivalent in respect to their identifiers and labels. Our case studies show that functional IncA is expressive and easy to use. Early performance measurements indicate that reusing established Datalog solvers yields more efficient execution times.

Practically speaking, we consider functional IncA to be a stepping stone for the compilation of other languages to Datalog. On one hand, our encoding paves the road for transferring years of research on functional programming languages to Datalog. For example, we show in this paper how standard defunctionalization [18] can be used to add first-class functions and first-class relations to functional IncA. Defunctionalization translates first-class functions to first-order functions and algebraic data types, which we can then compile to Datalog. On the other hand, we believe that our methodology for supporting user-defined functions and user-defined data types can be used to compile domain-specific languages to Datalog. We leave this avenue of research for future work.

In summary, we present the following contributions:

- We identify 5 principles that are necessary for the semantics-preserving translation of first-order functions to Datalog. We define the translation formally and adapt a demand transformation. This constitutes the first version of functional IncA (Section 3).
- We show how to compile user-defined algebraic data types to Datalog and extend functional IncA accordingly (Section 4).
- We add sets and set operations to functional IncA, extend the translation, and show how standard defunctionalization can be used to add first-class functions and first-class relations (Section 6).
- We demonstrate the expressiveness and ease of use of functional IncA by implementing a type checker, program transformation, and interpreter for the lambda calculus (Section 5), data-flow analyses for the While language (Subsection 7.1), and clone detection of Java bytecode (Subsection 7.2).
- We provide two backends for functional IncA, one targeting the incremental Datalog solver used by IncA, the other targeting the non-incremental Datalog solver Soufflé (Section 8).

2 Datalog Frontends: State of the Art

We are by far not the first to recognize the shortcomings of Datalog's frontend. Two opposing approaches have been explored in prior work to improve the expressiveness and/or usability of Datalog. We call these approaches *backend-first* and *frontend-first* and discuss them below.

Backend-first approach. The backend-first approach uses existing Datalog solvers as a starting point and extends them with new language features. Usually, extensions considered in the backend-first approach aim to increase the expressivity of Datalog, but sometimes also focus on usability. The backend-first approach has a long tradition in Datalog solvers and some features have become standard nowadays. For example, Datalog solvers support stratified negation and arithmetic operations, even though neither is part of core Datalog [15].

Modern Datalog solvers provide a range of different extensions that their users can choose from. For example, Soufflé [19] provides records, algebraic data types, and user-defined functions; Viatra Query [29], the Datalog solver used by IncA, supports user-defined data types and recursive aggregation over user-defined functions [22, 23]. While all of these features improve the frontend and make Datalog programming easier, the core language design remains the same: Horn clauses.

Horn clauses (a_0 `:-` $a_1, \ldots, a_n$) encode implications ($a_1 \wedge \ldots \wedge a_n \rightarrow a_0$). We argue Horn clauses are inadequate as a programming language, since they inhibit structured programming and enforce a flat structure. For example, a nested function call `res = f(g(h(x)))` becomes:

```
R(x, res) :- h(x, y), g(y, z), f(z, res)
```

That is, we must flatten the call chain. Or consider an expression that contains nested conditionals `(if (b1) x1 else x2) + (if (b2) x3 else x4)`, which becomes 4 separate Horn clauses:

```
R(b1, b2, x1, x2, x3, x4, res) :-  b1,  b2, res = x1 + x3.
R(b1, b2, x1, x2, x3, x4, res) :-  b1, !b2, res = x1 + x4.
R(b1, b2, x1, x2, x3, x4, res) :- !b1,  b2, res = x2 + x3.
R(b1, b2, x1, x2, x3, x4, res) :- !b1, !b2, res = x2 + x4.
```

These encodings are cumbersome to work with; they make programming and maintenance unnecessarily difficult. We would much rather use functional programming as a frontend.

While the backend-first approach does not fundamentally improve Datalog's frontend, it has one decisive advantage: It leverages existing solvers. These solvers are often the result of years of research and engineering. They automatically optimize Datalog programs, employ highly optimized data structures and algorithms, support profiling and debugging, provide incremental execution, and more. When designing new Datalog frontends, we should aim to reuse these systems. However, the state of the art moves in another direction.

Frontend-first approach. Quite a few recent research projects try to improve the frontend of Datalog by designing new DSLs to be used in its stead. We call these approaches frontend-first because the newly designed frontend is their starting point. In particular, frontend-first approaches do not build on top of an existing solver but develop a new solver specific to the newly designed frontend. This allows for great flexibility in the frontend's design.

For example, Flix [14] provides a Datalog frontend extended with lattices and monotonic functions. Flix embeds its Datalog frontend into a functional programming language, where constraints are first-class and can be generated at run time [13]. Formulog [9] provides a Datalog frontend extended with a data type for constructing SMT formulas and a constraint for solving them. While Formulog constraints are not first-class, the Datalog frontend is also embedded into a functional programming language. In both Flix and Formulog, the Datalog constraints can invoke functional code to assert a property or to construct new terms. In Formulog, functional code can also recursively query Datalog relations. While both systems present interesting designs, they also both implement their own Datalog solvers and do not benefit from prior engineering efforts.

Datafun [6] proposes a more drastic redesign for Datalog, namely as a higher-order functional programming language with fixpoint semantics. Datafun functions can accept and produce relations and the language supports the aggregation over lattices. As such, we believe Datafun's frontend is a well-suited replacement for Datalog. However, there are two limiting factors, First, in contrast to other modern implementations of Datalog, Datafun programs are constructor-free and enforce termination. While this equips Datafun with a nicer theory, it is a practical limitation, although one that could be easily eliminated. Second, like Flix and Formulog above, Datafun provides its own Datalog solver and existing optimizations and advances in Datalog engines have to be retrofitted to Datafun. For example, semi-naïve evaluation had to be adapted for Datafun [5], even though it has been the standard bottom-up evaluation model for a long time [27].

Our approach: Frontend compilation. We would like to achieve the best of both prior approaches: Build on top of existing Datalog solvers as in the backend-first approach, but be free to design functional and domain-specific frontends as in the frontend-first approach. The solution to this problem is compilation: By compiling the frontend language to Datalog, we can use existing solvers to run programs. This way, Datalog really becomes the intermediate representation (IR) of a compiler framework, where different Datalog frontends all generate the same Datalog IR. This architecture is well-known from existing compiler frameworks such as LLVM; we propose to adopt it for Datalog.

Although frontend compilation may seem like the obvious solution, it is difficult to implement. The problem is that Datalog imposes severe restrictions on programs, so that bottom-up evaluation is well-defined and terminates. When generating Datalog code, we must adhere to these restrictions. In the remainder of this paper, we show how a first-order functional language (Section 3) with algebraic data types (Section 4), and sets (Section 6) can be compiled to Datalog. In doing so, we will solve key challenges regarding user-defined functions and user-defined data types that can be transferred to other frontends.

3 Compiling First-Order Functions to Datalog

We want to provide a functional-programming frontend for Datalog. In this section, we tackle the first step in this direction: Compiling user-defined first-order functions to Datalog. While we already outlined why this is challenging in the introduction, here we revisit the problem with a more involved example before presenting our solution.

3.1 Compilation by example

In this paper and in our implementation, we use a simple functional frontend language that features first-order function definitions, let bindings, conditionals, and arithmetic operations. We also support algebraic data types, set operations, and first-class functions, which we will explain later. Consider the following recursive factorial function in functional IncA:

```
def fact(n: Int): Int = if (n == 0) 1 else n * fact(n - 1)
```

We aim to write functions like this and compile them to Datalog, so that we can use them as part of larger Datalog programs. A simple strategy gets us close to the desired result:

Principle 1: Functions as relations. It is well-known that functions `f : (T1, ..., Tn) -> T` can be encoded as relations `f : (T1, ..., Tn, T)`. We use this encoding of functions.

Principle 2: Control-flow paths as rules. For each path from function entry to function exit, we generate a rule that describes how inputs translates to outputs. Since control-flow paths are mutually exclusive in deterministic languages, so are the rules we generate.

When we apply this strategy to our factorial function, we obtain a relation `fact: (Int, Int)`. Since the `fact` function has two exits, we derive two rules that collect all conditions and computations along the path from entry to exit. In doing so, we introduce auxiliary variables for intermediate results as needed.

```
fact(n, out) :- n = 0, out = 1.
fact(n, out) :- n != 0, fact(n-1, out'), out = n * out'.
```

Unfortunately, like in the introduction, the Datalog rules violate *range-restrictedness*. A rule is range-restricted if every variable that occurs in the head of the rule is bound in the body of the rule. Range-restrictedness is an important property for Datalog programs and a prerequisite for bottom-up evaluation.Datalog engines like Soufflé [20] apply bottom-up evaluation to exhaustively enumerate all derivable tuples. Usually, this is an efficient

evaluation strategy, but it diverges for rules that are not range-restricted. In our example, the second rule is not range-restricted because `n` is not bound in the body, hence `n` could be any integer term. It follows that the `fact` relation contains infinitely many tuples. Therefore, Soufflé will reject the Datalog code we generated for the `fact` function.

It is hardly surprising that functions over (virtually) infinite domains describe (virtually) infinite relations. So is this approach doomed? To move forward, we make an important observation: Even though a function may be defined over an infinite domain, *any terminating application of that function will only see finitely many inputs.* If we can restrict a function's relation to these inputs, the entire relation turns finite and each rule becomes range-restricted.

To determine the relevant inputs of a function, we must consider how the function is used and what inputs it is applied to. For our factorial example, consider a main call `fact(5)`, which stipulates that `n = 5` is a relevant input of the `fact` relation. But since `fact` is recursive, we must also track which relevant inputs are induced by `n = 5`. If we collect all relevant inputs in `fact_input = {5,4,3,2,1,0}`, we can use this relation to guard the bodies of `fact`:

```
run_fact(out) :- fact(5,out).
fact(n, out) :- fact_input(n), n = 0, out = 1.
fact(n, out) :- fact_input(n), n != 0, fact(n-1, out'), out = n * out'.
```

Note how all rules are range-restricted now. Input variables are range-restricted by the query of the input relation; output variables are range-restricted because they are functionally dependent on the input variables. Thus, `fact` is finite when `fact_input` is finite.

Principle 3: Input relations as guards. For each function, collect all relevant inputs in an input relation and use the input relation as a guard for the function's relation.

Relevant inputs stem from external calls of the function or from recursive calls. Therefore, it is not easy to collect all relevant inputs in a relation. Fortunately, we can apply an existing algorithm that is well-known in the Datalog community: the magic-set transformation [8]. The magic-set transformation was developed to optimize the bottom-up evaluation of terminating Datalog programs. The key idea of the magic-set transformation is to only derive those tuples bottom-up that would also be derived by top-down evaluation, where the relevant inputs are known. To this end, the magic-set transformation generates Datalog rules for auxiliary relations that prescribe which inputs are relevant. Note that we say "inputs" here because the relations we care about correspond to functions; in general, the magic-set transformation collects terms that are known at the call-site during run time. Since function inputs are always known at the call-site during run time, the magic-set transformation will at least collect all relevant function inputs. Technically, we apply a more efficient variation of the magic-set transformation called the *demand transformation* [25] and we use that name in the remainder of the paper.

Principle 4: Demand transformation yields input relations. The demand transformation identifies all relevant inputs for each function in the program. Since all function call-sites must be known, our compilation strategy is not modular but requires the whole program.

For our example, the demand transformation will generate the following input relation:

```
fact_input(5).
fact_input(n-1) :- fact_input(n), n != 0.
```

We obtain one rule for each call of `fact`. The first rule collects the input of the main invocation `fact(5)`. The second rule collects the input of the recursive invocation and contains all constraints leading up to the call. Together, these two rules describe the required relation `fact_input = {5,4,3,2,1,0}`. Since `fact_input` is finite, `fact` is finite and contains the following tuples: `fact = {(5,120), (4,24), (3,6), (2,2), (1,1), (0,1)}`.

$$
\begin{array}{lll}
\text{(Functional programs)} & p & ::= \overline{F} \\
\text{(functions)} & F & ::= [\texttt{@main}]\ \texttt{def}\ f(\overline{x : T}) : T = e \\
\text{(expressions)} & e & ::= v \mid x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ (e)\ e\ \mathsf{else}\ e \mid f(\overline{e}) \mid \varphi(\overline{e}) \\
\text{(values)} & v & ::= \mathsf{base} \\
\text{(types)} & T & ::= \mathsf{Base}
\end{array}
$$

Figure 3 Functional IncA with first-order functions, base values, and base functions φ.

$$
\begin{array}{lll}
\text{(Datalog programs)} & D & ::= \overline{r} \\
\text{(rules)} & r & ::= \mathtt{R}(\overline{t})\ \texttt{:-}\ \overline{a}. \\
\text{(atoms)} & a & ::= t = t \mid \mathtt{R}(\overline{t}) \\
\text{(terms)} & t & ::= v \mid x \mid \varphi(\overline{t}) \\
\text{(values)} & v & ::= \mathsf{base}
\end{array}
$$

Figure 4 An intermediate representation for Datalog with base values and base functions.

So far, all function inputs were statically known. But we can easily extend our compilation strategy to support user-provided inputs. To this end, functional IncA allows the declaration of main functions:

```
@main def run_fact(n: Int): Int = fact(n)
```

The demand transformation will correctly propagate the input of `run_fact` to `fact`:

```
fact_input(n) :- run_fact_input(n).
fact_input(n-1) :- fact_input(n), n != 0.
```

But what is the input of `run_fact`? The input of `run_fact` is dynamic and must be provided by the user of the program. In Datalog, such data lives in the so-called extensional database, which is filled by the user prior to Datalog execution. We modify the demand transformation to generate a query of the extensional database for main functions.

Principle 5: Main input in extensional database. For each main function, we add a rule to the input relation that retrieves dynamic inputs from the extensional database.

For our example, we obtain the following input relation for `run_fact`:

```
run_fact_input(n) :- ext_run_fact_input(n).
```

The user can provide any number of inputs to `run_fact` as part of the extensional database. The Datalog engine will propagate those inputs to `run_fact_input` and fill all relations.

Note that our encoding retains crucial Datalog behavior, such as memoization and reuse. For example, consider we want to run `fact` on multiple inputs `5`, `7`, and `9`, all of which we put into the extensional database. How many tuples will relation `fact` contain? Since queries of `fact` will retrieve existing tuples when possible, the three `fact` computations will share all intermediate results and `fact` will only contain 10 tuples (the largest input plus one). A similar effect can be observed for functions like Fibonacci, where recursive calls can share results. All of this is transparent to the user.

3.2 Translating functional programs to Datalog, technically

We now implement Principles 1 and 2 from the previous subsection, that is, we translate functional programs to Datalog. In the subsequent subsection, we will explain and apply the demand transformation to implement the remaining principles.

$$
\begin{aligned}
[\![.]\!] :\ & e \to \mathcal{P}(t \times \mathcal{P}(a)) \\
[\![v]\!] =\ & \{(v,\ \emptyset)\} \\
[\![x]\!] =\ & \{(x,\ \emptyset)\} \\
[\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2]\!] =\ & \{(t_2, \{x = t_1\} \cup a_1 \cup a_2) \mid (t_1, a_1) \in [\![e_1]\!], (t_2, a_2) \in [\![e_2]\!]\} \\
[\![\mathsf{if}\ (e_1)\, e_2 \,\mathsf{else}\, e_3]\!] =\ & \{(t_2, \{t_1 = \mathsf{true}\} \cup a_1 \cup a_2) \mid (t_1, a_1) \in [\![e_1]\!], (t_2, a_2) \in [\![e_2]\!]\} \\
\cup\ & \{(t_3, \{t_1 = \mathsf{false}\} \cup a_1 \cup a_3) \mid (t_1, a_1) \in [\![e_1]\!], (t_3, a_3) \in [\![e_3]\!]\} \\
[\![f(e_1, \ldots, e_n)]\!] =\ & \{(y, \{f(t_1, \ldots, t_n, y)\} \cup a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in [\![e_1]\!], \ldots, (t_n, a_n) \in [\![e_n]\!]\} \\
& \text{where } y \text{ is fresh} \\
[\![\varphi(e_1, \ldots, e_n)]\!] =\ & \{(\varphi(t_1, \ldots, t_n), a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in [\![e_1]\!], \ldots, (t_n, a_n) \in [\![e_n]\!]\}
\end{aligned}
$$

Figure 5 Compiling expressions yields a set of alternative terms, each guarded by constraints.

$$
\begin{aligned}
[\![\mathtt{def}\ f(\overline{x : T}) : T' = e]\!]_{\mathit{fun}} =\ & \{\mathtt{f}(\overline{x}, y)\ \mathtt{:-}\ a,\ y = t. \mid (t, a) \in [\![e]\!]\} \quad \text{where } y \text{ is fresh} \\
[\![\overline{F}]\!]_{\mathit{prog}} =\ & \textstyle\bigcup_{f \in \overline{F}}\ [\![f]\!]_{\mathit{fun}}
\end{aligned}
$$

Figure 6 Compiling functions to Datalog rules.

Figure 3 defines the syntax of functional IncA. The language consists of first-order functions, let bindings, conditionals, and function calls. We distinguish calls to user-defined functions f from calls to base functions φ. Our compilation target is an intermediate representation (IR) of Datalog extended with base values and base functions as shown in Figure 4. This Datalog IR is compatible with many existing Datalog solvers, which support different kind of base functions. Note that we excluded negation from the Datalog IR because our translation does not require it.

We first translate expressions to Datalog. While an expression is structured and eventually computes a value, Datalog only provides flat terms. Thus, a nested expression $f(g(x))$ must be compiled to a flat term that is guarded by constraints $(y_2, \{g(x, y_1), f(y_1, y_2)\})$. Since conditional expressions $(\mathsf{if}\ (b)\ f(x)\ \mathsf{else}\ g(x))$ yield alternative values depending on b, compilation in general yields a set of alternative terms $\{(y_1, \{b = \mathsf{true}, f(x, y_1)\}), (y_2, \{b = \mathsf{false}, g(x, y_2)\})\}$. This correponds to Principle 2 from the previous subsection.

Figure 5 defines the translation of expressions as a compositional function $[\![.]\!]$. Values v and variables x directly translate to Datalog values and variables. Let bindings yield the body's result under a constraint that binds the let-bound variable. Conditionals compile to two alternative sets of terms: If the condition is true, the resulting terms are taken from the *then*-branch, otherwise they are taken from the *else*-branch. Calls to user-defined functions f translate to queries of a relation of the same name f, which has the function's result as an additional column in accordance with Principle 1. In contrast, calls to base functions φ translate to a call of the same function, but passing Datalog terms as arguments.

We use the translation of expressions $[\![.]\!]$ to compile function definitions $[\![.]\!]_{\mathit{fun}}$ and programs $[\![.]\!]_{\mathit{prog}}$ as shown in Figure 6. For a function definition, we compile its body and generate a separate Datalog rule for each alternative term that the body can yield. The constraints a of the term become constraints in the generated rule. To compile a whole program, we simply compile each function and collect the resulting rules.

For a concrete example, consider the translation of the Fibonacci function to Datalog:

```
⟦def fib(n) = if (n<2) n else fib(n-1) + fib(n-2)⟧ = { fib(n, y3) :- n<2 = true, y3 = n.,
  fib(n, y4) :- n<2 = false, fib(n-1,y1), fib(n-2,y2), y4 = y1+y2. }
```

The Fibonacci function compiles to two Datalog rules, one for the base case and one for the recursive case. But is this translation correct?

Translation correctness. We claim that our translation preserves the semantics of the original functional program. More precisely, we claim that if a function call $f(\overline{v})$ evaluates to w, then the generated Datalog program will also provide w as the only result of the call *under top-down evaluation*. Here we must require top-down evaluation for Datalog, since the generated rules are not necessarily range-restricted yet, which we will fix in the next subsection. Top-down evaluation is possible nonetheless, because it only explores required results and uses known values in doing so. Since the values of function arguments are always known during evaluation, top-down evaluation of the generated Datalog closely corresponds to function evaluation. However, we did not formalize top-down evaluation and therefore formulate translation correctness as a conjecture:

▶ **Conjecture 1** (Translation correctness). *Given a functional program p with a main function f such that $f(\overline{v})$ evaluates to w. Then the top-down evaluation of the Datalog atom $f(\overline{v}, x)$ under $[\![p]\!]_{prog}$ yields a single substitution $\{x \mapsto w\}$.*

A key component for proving this conjecture is to ensure the Datalog constraints behave deterministically, just like the original expression did:

▶ **Lemma 2** (Deterministic atoms). *Given an expression e such that $[\![e]\!] = \{(t_1, \overline{a_1}), \ldots, (t_n, \overline{a_n})\}$ both of the following hold:*

i. $[\![e]\!]$ *yields at least one result:* $\overline{a_1} \vee \ldots \vee \overline{a_n}$

ii. $[\![e]\!]$ *yields at most one result:* $(\overline{a_i} \wedge \overline{a_j}) \rightarrow t_i = t_j$

Proof. By structural induction over e. The only interesting case are *if*-expressions, where $(t_1 = \mathsf{true})$ and $(t_1 = \mathsf{false})$ are mutually exclusive. ◀

▶ **Lemma 3** (Deterministic rules). *Given f such that $[\![f]\!]_{fun} = \{f(\overline{x}, y_1) \text{:-}\ \overline{a_1}., \ldots, f(\overline{x}, y_n) \text{:-}\ \overline{a_n}.\}$ both of the following hold:*

i. $[\![f]\!]_{fun}$ *yields at least one result:* $\overline{a_1} \vee \ldots \vee \overline{a_n}$

ii. $[\![f]\!]_{fun}$ *yields at most one result:* $(\overline{a_i} \wedge \overline{a_j}) \rightarrow y_i = y_j$

Proof. Follows from Lemma 2. ◀

Note that Conjecture 1 does not make any assertions about non-terminating function calls. Indeed, some diverging functions compile to terminating Datalog programs. For example, `def f(x) = f(x)` compiles to `f(x,y) :- f(x,y)`. While function call `f(1)` diverges, query `f(1, y)` terminates and yields the empty substitution. However, Conjecture 1 ensures terminating function calls translate to terminating Datalog programs under top-down evaluation.

3.3 Demand-driven bottom-up evaluation

We compile functional programs to Datalog rules that execute well in top-down fashion, but may diverge under bottom-up evaluation. In bottom-up evaluation, Datalog solvers exhaustively enumerate all derivable tuples, starting from known facts. For example, the bottom-up evaluation of the factorial function will start with `fact(0,1)`, from which it can

derive `fact(1,1)`, `fact(2,2)`, `fact(3,6)`, `fact(4,24)`, and so on. This enumeration will not terminate, because bottom-up evaluation is unaware of the context in which relation `fact` is being used. Accordingly, we cannot apply any of the efficient Datalog solvers that use bottom-up evaluation, such as Soufflé.

The demand transformation by Tekle and Liu rewrites Datalog rules such that bottom-up evaluation becomes demand-driven and only computes tuples that are transitively demanded by the main query [25]. Indeed, bottom-up evaluation of the rewritten Datalog rules computes *exactly the same* tuples as top-down evaluation. Since we already asserted that top-down evaluation computes the correct result for terminating functional programs, the demand transformation allows us to apply bottom-up evaluation, also yielding the correct result.

We adopt the demand transformation, which transforms a set of Datalog rules in three steps: compute demand patterns, introduce demand predicates, derive demand rules. In this section, we replace the first step of the demand transformation to take functional IncA into account, adopt the second step unchanged, and extend the third step to account for the inputs of main functions. Later sections will make further changes.

Step 1. We compute demand patterns $\langle g, s\rangle$, where g is the name of a relation and $s \in (b\,|\,f)^*$ is a pattern string that indicates how the relation is queried, namely if an argument occurs bound or free. For functions, demand patterns can be easily computed by finding all function calls reachable from the main functions. Formally, given a functional program p, the demand patterns $dp(p)$ of p is the smallest set such that:

- For each main function (`@main def` $g(x_1, \ldots, x_n) = \ldots$) in p, we have $\langle g, b^n f\rangle \in dp(p)$. That is, main functions have demand with n bound parameters and one free return value.
- If demand pattern $\langle g, s\rangle \in dp(p)$ and g is defined as (`def` $g(\ldots) = e$) in p, we have $\langle h, b^n f\rangle \in dp(p)$ for each call $h(e_1, \ldots, e_n)$ in e.

The second and third step of the demand transformation operate on and rewrite the generated Datalog rules $D = [\![p]\!]_{prog}$. In particular, we will make no assumptions about the format of pattern strings s, so that we can later introduce extensions of Step 1 easily.

Step 2. We introduce demand predicates as guards into existing rules to implement Principle 3 from Subsection 3.1. Formally, we obtain a rewritten Datalog program $guarded(D)$:

- For each $\langle g, s\rangle \in dp(p)$ and each $(g(t_1, \ldots, t_m)$ `:-` $a_1, \ldots, a_n.)$ in D, we obtain a rule

  $$g(t_1, \ldots, t_m) \text{ :- } g_{\scriptstyle input}_s(t_1, \ldots, t_m|_s), a_1, \ldots, a_n.$$

  in $guarded(D)$, where $\bar{t}|_s$ selects those t_i that are bound according to pattern string s.

Note that the rules of unreachable functions are dropped and not propagated to $guarded(D)$.

Step 3. In the final step, we must derive those rules that define the input relations $g_{\scriptstyle input}_s$ to implement Principle 4 and Principle 5 from Subsection 3.1. Formally, we obtain a rewritten Datalog program $demanded(D)$ from $guarded(D)$ and the original program p as follows:

- We retain each rule from $guarded(D)$, such that $guarded(D) \subseteq demanded(D)$.
- For each main function (`@main def` $g(x_1, \ldots, x_n) = \ldots$) in p, we obtain a rule

  $$g_{\scriptstyle input}_s(x_1, \ldots, x_n) \text{ :- } ext_g_{\scriptstyle input}_s(x_1, \ldots, x_n).$$

  in $demanded(D)$, where $ext_g_{\scriptstyle input}_s$ is an extensional relation to be filled by the user. This implements Principle 5.

- For each rule $(g(\dots)$:- $a_1, \dots, a_n.)$ in *guarded*(D) and each $a_i = h(t_1, \dots, t_m)$, we obtain

  $$h_{\mathit{input}}_s(t_1, \dots, t_m|_s) \text{ :- } a_1, \dots, a_{i-1}$$

  to *demanded*(D), where s is the pattern string of $h(t_1, \dots, t_m)$, indicating which t_i are bound by the previous constraints $a_1, \dots, a_{i-1}$ already.

The demand transformation implements Principles 3 - 5 and ensures that the resulting Datalog derives the same tuples in bottom-up evaluation as in top-down fashion.

Example. To illustrate, consider again the Fibonacci function with a main call:

```
def fib(n) = if (n<2) n else fib(n-1) + fib(n-2)
@main def run(x: Int): Int = fib(x)
```

This program compiles to the following Datalog rules using the translation from Subsection 3.2:

```
fib(n, y3) :- n<2 = true, y3 = n.
fib(n, y4) :- n<2 = false, fib(n-1,y1), fib(n-2,y2), y4 = y1+y2.
run(x, y5) :- fib(x, y5).
```

We now apply our demand transformation. First, we derive demand patterns of the program, which are $\langle$`run`, $bf\rangle$ and $\langle$`fib`, $bf\rangle$. Note that all three calls of `fib` yield the same demand pattern. Second, we insert demand predicates into the rules according to the demand patterns:

```
fib(n, y3) :- fib_input_bf(n), n<2 = true, y3 = n.
fib(n, y4) :- fib_input_bf(n), n<2 = false, fib(n-1,y1), fib(n-2,y2), y4 = y1+y2.
run(x, y5) :- run_input_bf(x), fib(x, y5).
```

Third, to these rules we add the following rules to define the input relations:

```
run_input_bf(x) :- ext_run_input_bf(x).
fib_input_bf(x) :- run_input_bf(x).
fib_input_bf(n-1) :- fib_input_bf(n), n<2 = false.
fib_input_bf(n-2) :- fib_input_bf(n), n<2 = false, fib(n-1,y1).
```

The first and second rule are due to the main function `run`, which receives its input from the user and propagates it to `fib`. The third and fourth rule are due to the recursive calls of `fib`. Note how we retain all constraints prior to a call. In particular, we retain the first recursive call of `fib` as a constraint for the second recursive call of `fib`, although a smart compiler might eliminate this constraint subsequently. The resulting Datalog program is demand-driven and can be executed by standard bottom-up Datalog solvers.

Correctness. The demand transformation yields a Datalog program that derives the exact same tuples as a top-down evaluation [25]. As of Conjecture 1, top-down evaluation yields the correct tuples. Hence, so does bottom-up evaluation of the demand-driven Datalog rules:

▶ **Corollary 4** (Bottom-up translation correctness). *Given a functional program p with a main function f such that $f(\overline{v})$ evaluates to w. Then the bottom-up evaluation of the Datalog program demanded$(⟦p⟧_{prog})$ yields a database in which the query $f(\overline{v}, x)$ has a single match $\{f(\overline{v}, w)\}$.*

4 Compiling Algebraic Data Types to Datalog

The functional IncA we presented in the previous section supports user-defined functions ranging over base types. In this section, we explore how to extend functional IncA to allow user-defined data types. In particular, we want to faithfully compile recursive functions over algebraic data types to Datalog rules that existing bottom-up Datalog solvers can execute.

4.1 Compiling user-defined data types by example

We extend functional IncA to allow recursive definitions of user-defined algebraic data types, constructor calls, and pattern matching. As a simple example, consider the Peano numbers:

```
data Nat = Zero() | Succ(Nat)
def plus(m: Nat, n: Nat): Nat = m match {
  case Zero() => n
  case Succ(pred) => Succ(plus(pred, n)) }
@main def twice(n: Nat): Nat = plus(n, n)
```

We generate three kind of relations for an algebraic data type:

- **Constructor relations** represent the constructor functions of algebraic data types. We translate constructor calls in the program to queries of constructor relations, similar to how we translated regular function calls. In doing so, it is crucial we ensure only finitely many values are constructed during bottom-up evaluation of the resulting Datalog code.
- **Selector relations** map a constructed value to its constituents. We use selector relations to implement pattern matching. Importantly, queries of selector relations may never lead to the construction of new values.
- **Instance relations** enumerate all constructed instances of a data type. They will become useful when we introduce relational programming in Section 6.

To construct user-defined data at run time, we extend the Datalog IR with a built-in constructor `#constr` for each constructor `constr`. For example, `#Succ(#Succ(#Zero()))` encodes two as a Peano number. In practice, there are different ways a Datalog solver can support such built-in constructors. For example, we can define a generic built-in function that creates a new value given the constructor's name and arguments. We have used this approach in our implementation using IncA, but this would work in any Datalog solver that supports user-defined built-in functions, including Soufflé, Flix, and Formulog. Alternatively, if a Datalog solver natively supports algebraic data types, we can use their constructors directly or encode them using a number representation. For example, Soufflé supports algebraic data types (but not recursive functions over them) and we can generate a Soufflé data type and use its constructors. This is to say that adding built-in constructors to the Datalog IR does not limit the applicability of our approach in practice. Flix, Formulog, IncA and Soufflé have support for algebraic data. However, they do not support enumerating all instances of a specific algebraic data type like functional IncA. We will see how to enumerate all instances of an algebraic data type by utilizing instance relations in Section 6.

For the Peano numbers, we derive the following Datalog rules initially:

```
// constructor relations
Zero(n) :- n = #Zero().
Succ(p, n) :- n = #Succ(p).
```

```
// selector relations
un_Zero(n) :- Zero(n).
un_Succ(n, p) :- Succ(p, n).
```

```
// instance relation
Nat(n) :- Zero(n).
Nat(n) :- Succ(_, n).
```

Note that the rule of the `Succ` constructor relation is not range-restricted and consequently cannot be computed bottom-up. However, the rules of the selector and instance relations merely query the constructor relations. Hence, if we can ensure the constructor relations remain finite, all three kind of relations will be finite.

Like in the previous section, we seek to apply the demand transformation in order to track the demand of constructor relations. However, we need to adapt the demand transformation to account for our encoding of algebraic data types. Specifically, the constructor queries within the selector and instance relations must be ignored, since they do not actually indicate additional demand. Moreover, selector and instance relations do not require any rewriting themselves, because they merely query constructor relations to enumerate constructor tuples.

```
Zero(n) :- n = #Zero().   // no demand relation since there are no bound inputs
Succ(p, n) :- Succ_input_bf(p), n = #Succ(p).
Succ_input_bf(y4) :- plus_input_bbf(m, n), un_Succ(m, pred), plus(pred, n, y4).

// selector and instance relations un_Zero, un_Succ, and Nat as above
plus(m, n, out) :- plus_input_bbf(m, n), un_Zero(m), out = n.
plus(m, n, out) :- plus_input_bbf(m, n), un_Succ(m, pred),
                   plus(pred, n, y4), Succ(y4, y5), out=y5.
plus_input_bbf(n, n) :- twice_input_bf(n).
plus_input_bbf(pred, n) :- plus_input_bbf(m, n), un_Succ(m, pred).

twice(n, out) :- twice_input_bf(n), plus(n, n, out).
twice_input_bf(n) :- ext_twice_input_bf(n).

Zero(n) :- ext_Zero(n).
Succ(p, n) :- ext_Succ(p, n).
```

Figure 7 Compilation result for the `plus` and `twice` functions on Peano numbers.

Figure 7 shows the compilation result after demand transformation for the `plus` function on Peano numbers from above. Relation `Zero` has no demand relation because its demand pattern $\langle$`Zero`$, f\rangle$ does not specify bound inputs. Relation `Succ` has a demand relation `Succ_input_bf` that tracks the invocation of `Succ` in the recursive case of `plus`. Importantly, there is no demand on `Succ` from the selector or instance relations, as our adaption of the demand transformation will ensure. Relation `plus` shows how we compile pattern matching: Each case becomes an alternative rule that queries the selector. This is sufficient since we assume pattern matches are complete and overlap-free, so that their order does not matter.

Since `twice` is a main function, its demand relation queries an extensional input relation as described in the previous section. This way, users can for example request `twice(Succ(Zero()))`. But how can our Datalog program deconstruct the user-provided data? Recall that selector relations simply query constructor relations. Thus, we must include the user-provided algebraic data in our constructor relations. To this end, we require users to insert algebraic data in extensional constructor relations. We then generate one additional rule for each constructor that queries the corresponding extensional constructor relation, as shown at the end of Figure 7. We need to provide the contents of extensional constructor relations in the form of tuples consistent with the format supported by the targeted Datalog dialect. In the case of Soufflé, we insert tuples containing algebraic data and literal values of the Soufflé language in the extensional constructor relations.

4.2 Extending functional IncA with algebraic data types

Based on the observations from the previous subsection, we add algebraic data types to functional IncA. We then extend the translation from functional code to Datalog code and the demand transformation accordingly.

Figure 8 extends the abstract syntax of functional IncA with algebraic data types. For pattern matching we assume that patterns are complete and overlap-free. We do not change the syntax of Datalog since we model constructors as built-in functions φ.

We extend the translation of Subsection 3.2 from functional code to Datalog code to handle algebraic data types as shown in Figure 9. We add a new translation function $[\![.]\!]_{data}$ for data types and use that when compiling programs in $[\![.]\!]_{prog}$. The translation of functions

$$\begin{array}{llll}
\text{(Functional programs)} & prog & ::= & \overline{F}, \overline{d} \\
\text{(data definitions)} & d & ::= & \texttt{data}\ N = \overline{c(T, \ldots, T)} \\
\text{(expressions)} & e & ::= & \ldots \mid c(\overline{e}) \mid e\ \texttt{match}\ \{\overline{\texttt{case}\, c(x, \ldots, x) \texttt{=>} e}\} \\
\text{(types)} & T & ::= & \ldots \mid N
\end{array}$$

Figure 8 Extending the frontend syntax with algebraic data types.

$$[\![\overline{F}, \overline{d}]\!]_{prog} = \bigcup_{f \in \overline{F}} [\![f]\!]_{fun} \quad \cup \quad \bigcup_{d \in \overline{d}} [\![d]\!]_{data}$$

$$\begin{aligned}
[\![c(e_1, \ldots, e_n)]\!] &= \{(y, \{c(t_1, \ldots, t_n, y)\} \cup a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in [\![e_1]\!], \ldots, (t_n, a_n) \in [\![e_n]\!]\} \\
&\quad \text{where } y \text{ is fresh} \\
[\![e\ \texttt{match}\ \{\overline{cs}\}]\!] &= \bigcup_{(\texttt{case}\, c(\overline{x}) \texttt{=>} e') \in \overline{cs}} \{(t', \{\texttt{un_}c(t, \overline{x})\} \cup a \cup a') \mid (t, a) \in [\![e]\!], (t', a') \in [\![e']\!]\}
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{data}\ N = \overline{C}]\!]_{data} = &\ \{c(x_1, \ldots, x_n, y)\ \texttt{:-}\ y = \#c(x_1, \ldots, x_n). \mid c(T_1, \ldots, T_n) \in \overline{C}\} \\
&\cup \{c(x_1, \ldots, x_n, y)\ \texttt{:-}\ y = \mathit{ext_c}(x_1, \ldots, x_n, y). \mid c(T_1, \ldots, T_n) \in \overline{C}\} \\
&\cup \{\texttt{un_}c(y, x_1, \ldots, x_n)\ \texttt{:-}\ c(x_1, \ldots, x_n, y). \mid c(T_1, \ldots, T_n) \in \overline{C}\} \\
&\cup \{N(y)\ \texttt{:-}\ c(x_1, \ldots, x_n, y). \mid c(T_1, \ldots, T_n) \in \overline{C}\}
\end{aligned}$$

Figure 9 Translating algebraic data types to Datalog.

$[\![.]\!]_{fun}$ remains the same, but it uses an extended translation for expressions $[\![.]\!]$ that handles the new expressions: constructor calls and pattern matching. The translation of constructor calls is identical to the translation of regular function calls, except the generated code queries a constructor relation. Pattern matching yields alternative rules for each case, and each case queries the selector relation un_c to test if the term matches the pattern. The translation of data types $[\![.]\!]_{data}$ generates rules as described in the previous subsection: rules that invoke the built-in constructor functions, rules that query the extensional constructor relations, rules for the selector relations, and rules for the instance relations.

Next, we extend the demand transformation from Subsection 3.3 to consider constructors:

- In Step 1, when considering reachable subexpressions $h(e_1, \ldots, e_n)$, we also generate a demand pattern $\langle h, b \ldots bf \rangle$ when g is a constructor.
- In Step 2, note that selector and instance relations are never demanded, since we ignored them in Step 1. Hence, we propagate their rules unchanged to $\mathit{guarded}(D)$.
- In the last case of Step 3, we ignore atoms $a_i = h(t_1, \ldots, t_m)$ that occur in the rules of selector or instance relations. These atoms always query a constructor relation and we do not want to treat these queries as demand.

With these modifications, the demand transformation will correctly track the demand of constructors while ignoring selectors and instance relations. Together, the extended translation and the demand transformation constitute a compiler for functional IncA with algebraic data types. Since all rules of the generated Datalog code are range-restricted, we can run the code with off-the-shelf bottom-up Datalog solvers.

```
data Exp = Num(Int) | Lam(String, Type, Exp) | App(Exp, Exp) | Var(String)
data Type = TInt() | TFun(Type, Type)
data UExp = UNum(Int) | ULam(String, Exp) | UApp(Exp, Exp) | UVar(String)
def typeOf(ctx: Ctx, exp: Exp): Maybe[Type] = exp match {
  case App(fun, arg) => typeOf(ctx, fun) match {
    case Just(TFun(ty1, ty2)) => typeOf(ctx, arg) match {
        case Just(argty) => if (eqType(argty, ty1)) Just(ty2) else Nothing()
  ... }
def erase(exp: Exp): UExp = exp match {
  case Num(v) => UNum(v)
  case Lam(n, ty, b) => ULam(n, erase(b))
  case App(fun, arg) => UApp(erase(fun), erase(arg))
  case Var(n) => UVar(n) }
def interp(env: Env, exp: UExp): Maybe[Val] = exp match {
  case UApp(fun, arg) => interp(env, fun) match {
    case Just(VClosure(param, prog, fenv)) => interp(env, arg) match {
      case Just(argv) => interp(BindEnv(param, argv, fenv), body)
  ... }
@main def run(exp: Exp): Maybe[Val] = typeOf(EmptyCtx(), exp) match {
  case Just(ty) => interp(EmptyEnv(), erase(exp))
  case Nothing() => Nothing() }
```

Figure 10 A type checker, type erasure, and interpreter for a lambda calculus with numbers.

5 Case study: Type Checking, Type Erasure, and Interpretation

Functional IncA supports user-defined functions and data types. In this section, we demonstrate that these features allow us to express interesting computations in Datalog. In particular, we implement a type checker, type erasure, and an interpreter for a lambda calculus with numbers as illustrated in Figure 10. These functions compile to complex Datalog code that could not practically be written by hand.

Figure 10 shows an excerpt of the relevant data types and functions, all of which are completely standard. In particular, we describe the expressions of the simply typed lambda calculus `Exp` and the untyped lambda calculus `UExp` as algebraic data types. We define a type checker `typeOf` as a function in functional IncA, but only show the `App` case here. Our implementation supports parametric polymorphism by applying monomorphization before translating to Datalog. Since the `App` case has five alternative control-flow paths, this case alone compiles into five Datalog rules for `typeOf`. For example, consider the rule generated for the path that yields `Just(ty2)`:

```
typeOf(ctx, exp, out0) :-
  typeOf_input_bbf(ctx,exp), un_App(exp,fun,arg), typeOf(ctx,fun,o1),
  un_JustType(o1,funty), un_TFun(funty,ty1,ty2), typeOf(ctx,arg,o2),
  un_JustType(o2,argty), eqType(argty,ty1,o3), o3 == true, JustType(ty2,out0).
```

This Datalog rule consists of 10 atoms, where the selector predicates ensure that the correct control-flow path has been chosen. Overall, the `typeOf` function consists of 24 lines of code, but compiles to 114 lines of complex Datalog code with mutually dependent relations `typeOf` and `typeOf_input`. These numbers represent the Datalog program after applying optimizations. In contrast to program optimizations of functional and imperative programs, our Datalog optimizations reduce the number of rules and atoms instead of increasing them.

Next, we define type erasure as a transformation from `Exp` to `UExp`. Although function `erase` is completely standard, this is the first program transformation implemented in Datalog to the best of our knowledge. While `erase` is guaranteed to terminate, we can also define functions whose termination is undecidable. Specifically, we implement a standard interpreter `interp` for the untyped lambda calculus, which is a Turing-complete language. Indeed, the Datalog program is only guaranteed to terminate when the original interpreter terminates.

Overall, the type checker, type erasure, and interpreter comprise 8 algebraic data types and 7 functions. We compile this code to 65 relations defined by 154 rules that contain 484 atoms in total. These numbers are measured after optimization, where we eliminate aliases and propagate constants.
Although implementing an interpreter in Datalog may seem to be of little use, this and similar challenges occur during program analysis regularly. For example, Pacak et al. recently have shown how to compile typing rules to Datalog to derive incremental type checkers systematically [17]. They also mention that it is necessary to translate the dynamic semantics of a language to Datalog in order to support the incremental type checking of a dependently typed programming language. Similarly, data-flow analyses often need to abstractly interpret programs, for example, to determine the bounds of numeric variables or the value of a Boolean condition. Functional IncA can also support such data-flow analyses, but we must be able to express control-flow graphs and other relations.

6 Mixing Functions and Relations

The previous sections showed how we can use functional programming as a frontend for Datalog. However, in doing so, we have also lost a key feature of Datalog: relations. Indeed, functional IncA makes it difficult to encode non-functional relations, such as the edges of a graph. In the present section, we show how we can elegantly extend functional IncA to re-introduce relations.

6.1 Computing a control-flow graph functionally

Consider we want to compute the control-flow graph (CFG) of a program as part of a Datalog-based program analysis. We want to represent the CFG such that it corresponds to a Datalog relation, so that we can easily compute its transitive closure later. While the functions of functional IncA compile to Datalog relations, our functions cannot be used to encode arbitrary relations. In particular, a function (`def flow(from: Stm): Stm = e`) cannot handle conditional statements that fork the control flow and connect to multiple successor statements. To support such relations, we must extend our frontend language.

We want to extend functional IncA in a way that integrates functions and relations elegantly. This is a language-design challenge and therefore naturally somewhat subjective. But it is the reason why we rejected the first idea that came to mind: to introduce relations next to functions. For example, a top-level relation (`rel flow(from: Stm, to: Stm) :- constraints`) could capture the CFG of a program. The problem is that we are now back at constraint programming, which is exactly what we wanted to avoid with functional IncA.

We propose a different extension of functional IncA that not only avoids this problem but that is simpler too: We introduce sets and tuples. Immutable sets and tuples are staple ingredients of functional programming and programmers already know how to use them. Moreover, any relation can be encoded as a set containing tuples of related values. Thus, the only question is if and how we can map functional programs over sets and tuples to Datalog. But first, let us illustrate how the extended functional IncA can be used.

In their classic textbook, Nielson et al. [16] compute the control flow of a While-statement through three functions. We can represent these functions in the extended functional IncA almost verbatim as shown in Figure 11. Here, `init` is a regular function whereas `final` and `flow` compute sets. A set literal `{e1,...,en}` constructs a set and set union `++` composes two sets. For example, `final` uses these features to compute the final statement of each conditional

```
data Exp = ...
data Stm = Assign(String, Exp) | Sequence(Stm, Stm) | If(Exp, Stm, Stm) | While(Exp, Stm)
def init(stm: Stm): Stm = ... // a regular function that finds the statement's entry
def final(stm: Stm): Set[Stm] = stm match { // finds all of the statement's exits
  case Assign(x, a) => {stm}
  case Sequence(s1, s2) => final(s2)
  case If(b, s1, s2) => final(s1) ++ final(s2)
  case While(b, s) => {stm} }
// flow as seen in Figure 1 (Introduction)
```

Figure 11 Computing the control-flow graph as a set of tuples in out extended Datalog frontend.

(functions)	F	$::= \ldots \mid [\texttt{@main}]\ \texttt{def}\ f(\overline{x : T}) : \mathsf{Set}[T] = s$
(set expressions)	s	$::= \{\overline{e}\} \mid s \mathbin{++} s \mid \{e \mid \overline{pred}\} \mid \mathsf{let}\ x = e\ \mathsf{in}\ s \mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s \mid f(\overline{e})$
(predicates)	$pred$	$::= e \mid e\ \mathsf{in}\ s \mid e\ \mathsf{in}\ N$
(expressions)	e	$::= \ldots \mid \mathsf{fold}(f, f, f)$

Figure 12 Extended abstract syntax with set and set operations.

branch. Sets can be processed through set comprehensions as shown in the definition of `flow` which can be seen in Figure 1. In particular, `(x1,...,xn) in set` retrieves the elements of `set`, binds those `x` that are free, and tests for membership of those `x` that are bound.

Our encoding of relations makes it easy to implement computations that exercise Datalog's declarative fixpoint semantics, such as transitive closure, cycle detection, and recursive aggregation. We have already demonstrated such computations in the introduction of this paper and refrain from repeating them here. Instead, we show how to translate functional programs with sets and tuples to Datalog.

6.2 Translating tuples and first-order sets to Datalog

The translation of sets and tuples to Datalog is mostly straightforward except for one thing: neither sets nor tuples are first-class in Datalog. For tuples this is hardly an issue since we can simply flatten tuples when translating them to Datalog. For example, a function `foo(t: (T1,...,Tn)): (U1,...,Um)` becomes a flat relation `foo(T1,...,Tn,U1,...,Um)`, and a function call `foo(e)` becomes `foo(t1,...,tn,u1,...,um)`, where `e` translates to `n` terms `(t1,...,tn)` and the function call yields `m` result terms `(u1,...,um)`. Although our implementation supports tuples, we omit tuples from our translation semantics and focus on sets instead.

We want to translate sets to Datalog relations, but relations are first-order in Datalog and can only appear as top-level definitions. Thus, if we want to support first-class sets in functional IncA, we need to lift those sets first. For example, to translate a call `transitive({(1,2),(2,3),(3,4)})` to Datalog, we have to translate `{(1,2),(2,3),(3,4)}` to a top-level relation that can be queried from within `transitive`. To achieve this, we propose a clean solution in two steps:

1. We extend functional IncA first-order sets, which may only appear as function results. First-order sets translate to first-order relations as shown in the present subsection.
2. The subsequent subsection shows that a standard defunctionalization transformation simultaneously adds support for first-class functions and first-class sets to functional IncA.

Figure 12 defines the extended functional IncA, where we introduce first-order sets syntactically through a new non-terminal `s`. This syntactic differentiation does not replace type checking of the functional code, but serves to explain which expressions may yield

$$
\begin{aligned}
[\![\{\overline{e}\}]\!] &= \textstyle\bigcup_{e \in \overline{e}} [\![e]\!] \\
[\![s_1 \mathbin{++} s_2]\!] &= [\![s_1]\!] \cup [\![s_2]\!] \\
[\![\{e \mid p_1, \dots, p_n\}]\!] &= \{(t, \{t_1 = \mathsf{true}, \dots, t_n = \mathsf{true}\} \cup a \cup a_1 \cup \dots \cup a_n) \\
&\quad \mid (t, a) \in [\![e]\!], (t_1, a_1) \in [\![p_1]\!]_{pred}, \dots, (t_n, a_n) \in [\![p_n]\!]_{pred}\} \\
[\![\mathsf{fold}(f_{init}, f_{op}, f_{set})]\!] &= \{(\mathsf{aggregate}(f_{set}, \mathsf{toPrimitiveFun}(f_{init}), \mathsf{toPrimitiveFun}(f_{op})), \emptyset)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![e]\!]_{pred} &= [\![e]\!] \\
[\![e \ \mathsf{in}\ s]\!]_{pred} &= \{(\mathsf{true}, \{t_1 = t_2\} \cup a_1 \cup a_2 \mid (t_1, a_1) \in [\![e]\!], (t_2, a_2) \in [\![s]\!]\} \\
[\![e \ \mathsf{in}\ N]\!]_{pred} &= \{(\mathsf{true}, \{N(t)\} \cup a \mid (t, a) \in [\![e]\!]\}
\end{aligned}
$$

Figure 13 Compiling sets and set operations to Datalog.

sets without presenting functional IncA's type system, which is completely standard and uninteresting. First-order sets may only occur as the body of a function that yields a set and within other set expressions. A set comprehension can use predicates *pred* to check a boolean condition e, to query another set (e in s), or to query all instances of an algebraic data type (e in N). Here we finally see why we introduced instance relations for algebraic data types in Section 4. At last, we can convert a set to an atomic value through $\mathsf{fold}(f_{init}, f_{op}, f_{set})$, where f_{set} must be the name of a top-level definition.

We extend $[\![.]\!]$ to also handle set expressions s, and we add a translation function $[\![.]\!]_{pred}$ for predicates. Figure 13 shows both translation functions. A set literal translates to a set of alternative terms and set union computes the union of alternative terms. A set comprehension builds all terms t generated by e for which all predicates are true.

We can only translate folds if the targeted Datalog engine supports aggregation over user-defined functions. In our experience, such user-defined functions must be implemented in the same language as the Datalog engine (e.g., C++ for Soufflé, a JVM language for Formulog and IncA). Thus, fold operations are considered built-in functions φ by Datalog engines. We extend the Datalog IR with aggregation accordingly:

(Datalog terms) $\quad t \ ::= \ \dots \mid \mathsf{aggregate}(R, \varphi, \varphi)$

All we have left to do is to translate frontend functions f to built-in functions φ, which we assume function toPrimitiveFun accomplishes. In our implementation, we target IncA and compile user-defined frontend functions to Scala, which was straightforward. Soufflé does not support aggregation over user-defined functions, hence we cannot target Soufflé if the functional IncA program contains fold operations.

6.3 First-class functions and first-class sets

Functional IncA paves the road for transferring insights from functional programming languages to Datalog. Here, we exemplify this potential by studying defunctionalization in the context of functional IncA. Defunctionalization [18] is a well-known compilation technique that compiles higher-order functions into first-order functions and first-class function values into algebraic data. In particular, defunctionalization generates auxiliary *apply* functions that dispatch on the algebraic data to execute the corresponding function body. Since functional IncA supports first-order functions and algebraic data types, we can apply defunctionalization to extend functional IncA with first-class functions.

$$
\begin{array}{llll}
\text{(expressions)} & e & ::= & \ldots \mid f \mid (\overline{x : T}) \Rightarrow e \mid (\overline{x : T}) \Rightarrow s \mid e(\overline{e}) \\
\text{(types)} & T & ::= & \ldots \mid \overline{T} \Rightarrow T
\end{array}
$$

Figure 14 Adding first-class functions to our Datalog frontend.

Figure 14 shows how we extend functional IncA's syntax with first-class functions. A function value is either a reference to a top-level function f or a lambda. Note that we permit lambdas to yield sets, since they will translate to first-order functions, which we translate to first-order relations. Finally, we adapt function application to allow any expressions in function position.

For example, consider an excerpt from our data-flow analyses of the While language:

```
def findExps(exp: Exp, f: Exp => Boolean): Set[Exp] = (exp match {
  case Var(s) => {}
  case Num(i) => {}
  case Add(e1, e2) => findExps(e1, f) ++ findExps(e2, f)
}) ++ (if (f(exp)) {exp} else {})
def freevars(exp: Exp): Set[String] = {varName(e) | e in findExps(exp, isVar)}
def availableExps(exp: Exp): Set[Exp] = findExps(exp, (e: Exp) => e match {
  case Var(s) => false
  case Num(i) => false
  case Add(e1, e2) => true })
```

We define a higher-order function `findExps` that selects all subexpressions satisfying predicate `f`. We use `findExps` twice, once to find all free variables of an expression and once to find all non-trivial subexpressions. We implement a standard defunctionalization transformation that translates this program into a first-order functional program:

```
data Defun0 = Funref0() | Lambda0()
def applyDefun0(fun: Defun0, e: Exp): Boolean = fun match {
  case Funref0() => isVar(e)
  case Lambda0() => e match {
    case Var(s) => false
    case Num(i) => false
    case Add(e1, e2) => true } }
def findExps(exp: Exp, f: Defun0): Set[Exp] = (...) ++ (if (applyDefun0(f, exp)) {exp} else
    {})
def freevars(exp: Exp): Set[String] = {varName(e) | e in findExps(exp, Funref0())}
def availableExps(exp: Exp): Set[Exp] = findExps(exp, Lambda0())
```

We can then translate the defunctionalized program to Datalog as described before. Thus, we have successfully extended functional IncA with first-class functions.

But how does this enable first-class sets? We already added support for first-order sets, which may only occur as function results. But since first-class functions translate to first-order functions, first-class functions may also yield sets. Thus, we can encode a first-class set `s` as a thunk `() => s`. For example, we can define a higher-order relation `transitive` as follows:

```
def transitive(cfg: () => Set[(Stm, Stm)]): Set[(Stm, Stm)] =
  cfg() ++ {(s1,s3) | (s1,s2) in cfg(), (s2,s3) in transitive(cfg)}
@main def transitiveFlow(prog: Stm): Set[(Stm, Stm)] = let cfg = () => flow(prog) in
  transitive(cfg)
```

Since function values become algebraic data, thunk-encoded sets are truly first-class: They can be assigned to variables and they can be passed as arguments. This shows how functional IncA permits insights from functional programming languages to carry over to Datalog, where they can unleash additional benefits.

```
def gen(stm: Stm): Set[(String,Maybe[Stm])] = stm match {
  case Assign(x, a) => {(x, Just(stm))}
  case Sequence(s1, s2) => {}
  case If(c, s1, s2) => {}
  case While(c, s) => {} }
def retain(stm: Stm, x: String): Boolean = ...
def entry(stm: Stm, prog: Stm): Set[(String, Maybe[Stm])] =
  if (stm == init(prog))   {(x, Nothing()) | x in freevarsStm(prog)}
  else                     {(x,d) | (pred, stm) in flow(prog), (x,d) in exit(pred, prog)}
def exit(stm: Stm, prog: Stm): Set[(String,Maybe[Stm])] =
  gen(stm) ++ {(x,d) | (x,d) in entry(stm, prog), retain(stm, x)}
@main def allExits(prog: Stm): Set[(Stm, String, Maybe[Stm])] =
  {(s,x,d) | s in Stm, (x,d) in exit(s, prog)}
```

Figure 15 A reaching definitions analysis for the While language that we compile to Datalog.

7 Case Studies: Data-Flow Analyses and Clone Detection

We have presented functional Datalog frontend with relations that compiles to Datalog. In these final case studies, we want to demonstrate why this design is useful and how it enables a new way of implementing Datalog-based static analyses. To this end, we implemented flow-sensitive reaching definitions and interval analyses for the WHILE language in functional IncA. Additionally, we show how to describe clone detection of Java bytecode.

7.1 Data-Flow Analyses

Figure 15 shows an excerpt of the reaching definitions analysis, which determines where a variable was last defined. Our analysis implementation is completely standard except that we use a `retain` filter in place of the usual `kill` set. This is because functional IncA does not support negation yet, which is needed for set difference. We hope to extend functional IncA with negation in future work, but note that negation in Datalog is far from trivial and deserves a separate study.

The reaching definitions case study shows how we benefit from using functions and relations. The main benefit of functions is the ease of implementation in a well-known programming paradigm, as illustrated by `gen` in our example. The main benefit of relations is the implicit fixpoint semantics provided by Datalog. Specifically, note that `entry` and `exit` call each other unconditionally and diverges under functional-programming semantics. However, Datalog implicitly computes the least fixpoint of relations, which is computable because the relations are finite: There are only finitely many variables and assignments in any program. Here, functional IncA reaps the rewards of compiling to Datalog.

In the reaching definitions analysis, the fixpoint computation within `entry` and `exit` only invokes simple functions `gen` and `retain`. Therefore, it is reasonable to implement the reaching definitions in Datalog directly, although we believe functional IncA is easier to use. In contrast, our second data-flow analysis implements an interval analysis that requires complex functions to abstractly interpret expressions. We show an excerpt of the interval analysis in Figure 16 and Figure 2. We use data type `Val` to represent abstract values and use relations `entry_var` and `exit_var` to map variables to their abstract value. For an `Assign` statement, `exit_var` invokes an abstract interpreter `aeval` that computes the abstract value of the assigned expression. Even for this simple WHILE language, the abstract interpreter already consists of 90 lines of functional code that compile to 342 lines of Datalog code. Moreover, `aeval` is part of the fixpoint loop, because it invokes `entry_var` for variable references, which invokes

```
data Val = BotVal() | IntervalVal(Interval) | BoolVal(Bool) | TopVal()
...
// entry_var, exit_var, and aeval as shown in Figure 2 (Introduction)
def add(v1: Val, v2: Val): Val = ...
def addInterval(iv1: Interval, iv2: Interval): Interval = iv1 match {
  case TopInterval() => TopInterval()
  case IV(l1, h1) => iv2 match {
    case TopInterval() => TopInterval()
    case IV(l2, h2) => IV(l1 + l2, h1 + h2) } }
def joinVal(v1: Val, v2: Val): Val = ...
def joinInterval(iv1: Interval, iv2: Interval): Interval = iv1 match {
  case TopInterval() => TopInterval()
  case IV(l1, h1) => iv2 match {
    case TopInterval() => TopInterval()
    case IV(l2, h2) => widenInterval(IV(Math.min(l1, l2), Math.max(h1, h2))) } }
```

Figure 16 Interval analysis of the While language using abstract interpretation.

```
def getStm(inst: Instruction): Set[Stm] = {
  InvokeStm(recvExp, meth, args) |
    (inst, v) not in _AssignReturnValue,
    (inst, _, meth, recv, _) in _VirtualMethodInvocation,
    recvExp in getExp(recv),
    args in getArgs(inst, 0) } ++ ...
def getExp(v: String): Set[Exp] = ...
def getArgs(inst: Instruction, currentIdx: Int): Set[List[Exp]] = ...
def isStmClone(s1:Stm, s2:Stm, iPairs:NPairs, lPairs:NPairs): Boolean = (s1,s2) match {
    case (InvokeStm(r1, m1, a1), InvokeStm(r2, m2, a2)) =>
      m1 == m2 && isExpClone(r1, r2, iPairs) && isArgListClone(a1, a2, iPairs)
    ... }
```

Figure 17 Clone detection of Shimple Code.

`exit_var`, which invokes `aeval`. Therefore, `aeval` really must translate to Datalog rules and cannot be represented as a built-in function, because then it could not invoke `entry_var`. Finally, note that we use a user-defined function `joinVal` to aggregate abstract values in `entry_var`. In particular, `joinVal` implements widening on intervals to ensure the analysis always terminates. All of these concerns are easy to address in functional IncA, because we can use functional programming while relying on Datalog's fixpoint semantics.

7.2 Clone Detection

Figure 17 shows an excerpt of how to construct an abstract syntax tree of Shimple code and apply clone-detection techniques such as testing for alpha-equivalence. Shimple is a variant of the Java bytecode representation Jimple [28] in SSA form. To access the Shimple representation, we extend functional IncA to read Soufflé relations, because the Doop framework [10] generates Soufflé facts. Using Soufflé facts enables us to detect clones of real-world Java programs. The Soufflé relations are prefixed by an underscore. Technically, we compile the Soufflé program and the functional program to a single Datalog program. However, we do not derive demand patterns for relations of the Soufflé program.

The function `getStm` constructs an abstract syntax tree representation of Shimple code. We highlight the case for constructing an invocation statement. We only generate an invocation statement for an instruction `inst` if it is a virtual method invocation and the instruction does

not assign a return value for the given instruction. Note that functional IncA does not allow negation in general. However, it is possible to query Soufflé relations negatively as we do not apply the demand transformation to Soufflé relations. Hence, the demand transformation does not introduce negated dependency cycles. We generate the receiver of the method call by using function `getExp` which constructs an expression tree given a variable name. At last, we construct the argument of the given invocation statement by calling `getArgs`. Note while `getStm`, `getExp` and `getArgs` have `Set` as return type, the functions yield singleton sets. Returning a set is necessary due to the fact that we query Soufflé relations.

Next, we use the constructed abstract syntax trees as a basis to detect clones. We show a clone-detection function `isStmClone` which checks if the statements are alpha-equivalent. We traverse the statements `s1` and `s2` simultaneously while checking that the statements and inner expressions are equal. Because we rely on Soufflé relations generated by the Doop framework [10], we could integrate static analysis information such as points-to information into clone detection. The case study shows that describing alpha-equivalence of Java bytecode in a functional style is straightforward. It is possible to realize more sophisticated clone-detection techniques using functional IncA such as structural diffing [11].

8 Implementation and Performance Evaluation

In this section we will discuss our implementation and do an early performance evaluation.

8.1 Implementation

We implemented functional IncA by compiling it to a Datalog IR provided by the IncA framework. The Datalog IR can target two different backends namely IncA and Soufflé without any change to the underlying Datalog solvers VIATRA [29] and Soufflé [20] respectively. Our compiler generates Datalog code as shown in this paper, including the demand transformation. This implementation not only demonstrates the feasibility of our design, but also shows how advantageous it is to reuse existing Datalog solvers. In particular, the VIATRA Datalog solver supports incrementality: Changes in extensional relations trigger incremental updates in derived relations. We inherit this incrementality for free. For example, we can run the interval analysis of Subsection 7.1 incrementally by diffing the input programs and feeding the resulting patch to IncA [11]. Targeting Soufflé allows us to generate efficient and scalable C++ programs that run on multi-core machines. However, Soufflé does not support user-defined aggregation, hence we do not support translating functional IncA programs containing fold operations. The implementation is available at `https://gitlab.rlp.net/plmz/inca-scala`.

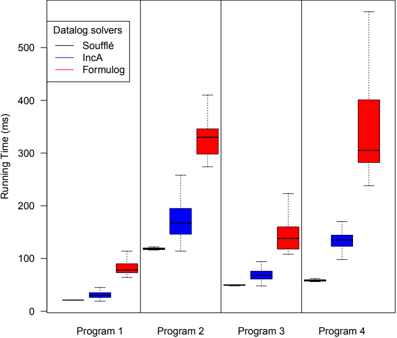
8.2 Performance Evaluation

We evaluate the performance of functional IncA and show that it is advantageous to use established solvers instead of implementing custom Datalog solvers for new frontends. We compare the running times of executing a data-flow analysis for the While language run with Souffé, IncA, and Formulog. We choose Soufflé and IncA as they are already established Datalog frameworks. We choose Formulog because it is one representative of the *frontend-first* approach which combines first-order ML functions with Datalog by implementing a custom Datalog solver. Even though IncA uses an incremental Datalog solver VIATRA [29], we do not measure the incremental performance of IncA which we leave as future work.

The data-flow analysis that we run is an adapted interval analysis. The analysis collects all integers $-100 \leq i \leq 100$, a variable can be assigned to. Whenever we encounter an integer $i < -100$ we return the default value -1000 and when we encounter an integer $i > 100$ we

return 1000. We implement this cut-off to ensure that the data-flow analysis terminates in the presence of loops. We have chosen this type of analysis instead of an interval analysis, because an interval analysis requires user-defined aggregation which Formulog and Soufflé currently do not support. We implement four different programs as input of the data-flow analysis. The programs consist of nested *while* and *if* statements and are designed in such a way that a lot of information has to propagated along the edges of the control-flow graph.

For Formulog and IncA, both of which are Datalog solvers that run on the JVM, we first do 10 warmup runs and then measure 90 runs. We do not measure the time it takes to initialize the extensional database but only measure the running times of deriving the intensional database. For Soufflé, we compile an executable and measure the running time of the compiled Soufflé solver to derive the intensional database 9 times. Note that we do no warmup for Soufflé programs as they are compiled to C++ and then to executable machine code. We store the extensional database within input files and do not measure the I/O actions needed to read those input files. We load the contents of the input files into RAM by executing the compiled Soufflé program once. Hence, the following measured runs access the extensional database stored in RAM. We performed our benchmarks on a machine with an Intel Core i7 at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 11.4, Java 1.14.0_1.



We show the running times of deriving the intensional database in milliseconds for each program in the figure above. We see that the custom Datalog solver for Formulog is slower than the established solvers such as Soufflé and IncA for all input programs. The Formulog solver is $\sim 3.7x$ slower than the Soufflé solver and $\sim 2.2x$ slower than the IncA solver. Note that the compiled executable of Soufflé has the fastest running time of all three solvers. This shows that is desirable to compile Datalog with functional constructs to already established Datalog dialects instead of implementing custom solvers for new Datalog frontends if possible.

9 Related Work

We propose functional programming with sets as a frontend for Datalog to replace Datalog's traditional constraint programming. This design differs from most prior works, which retain constraint programming as a basis and add functional aspects on top of it. Our approach has three advantages: (i) functional programming is easy to use, (ii) we can compute fixpoints across functions and relations, and (iii) we can reuse existing Datalog solvers. In the remainder of this section, we discuss related work.

IncA is an incremental Datalog framework that supports recursive aggregation over user-defined functions and data types [22, 23]. These user-defined functions and data types must be implemented in a JVM language and cannot query Datalog relations. The original

frontend of IncA provides a shallow abstraction over Datalog called *pattern functions* [24]. These pattern functions consist of sequences of constraints and really are not comparable to the functional programming we support in functional IncA.

Flix [14] exposes constraint programming to the user, but extends it with functional programming. The runtime system of Flix executes functional code but also contains a custom Datalog solver. While functional and Datalog aspects are intertwined in Flix, they cannot interact as tightly as the functions and relations in our approach. Specifically, user-defined functions cannot recursively query derived relations, as required by our interval analysis. However, it is also not obvious how to extend our approach to compile Flix to Datalog, because Flix supports the generation of additional Datalog constraints at run time [13].

Formulog [9] combines first-order ML functions with Datalog and SMT solvers. In particular, Datalog rules can contain ML expressions and ML code can recursively query Datalog relations. Formulog's runtime understands both languages, which is why a custom Datalog solver was needed that can evaluate ML expressions and Datalog constraints interleaved. Our approach should naturally extend to Formulog. Indeed, we could add SMT solving as a built-in function (`def solveSMT(spec: String): String`) and rely on user-defined data types for SMT formulae and models, both of which are built-in types in Formulog.

Datafun [6] defines a higher-order functional programming language with sets and fixpoint semantics. From a language-design perspective, Datafun is the most closely related work. Both languages support commonly known functional expressions. One difference is how fixpoint computations are expressed in the surface syntax. Datafun provides a fixpoint expression which explicitly states over which function a fixpoint will be computed. However, in functional IncA the fixpoint computation is not explicitly given but implicitly given by the dependencies between functions. Datafun and functional IncA follow different design philosophies. Datafun provides a termination guarantee: If a Datafun program is well-typed, then a unique least fixed point exists and the program will terminate. Our language does not provide such a guarantee since a well-typed program can still diverge. Consequently, Datafun is more restrictive to guarantee termination while functional IncA gives developers more freedom (and responsibility). Datafun requires that the lattice type over which a fixpoint is computed does not contain an infinite ascending chain. One disadvantage of Datafun's design is that some programs that terminate in our system are not accepted by Datafun. For example, the interval analysis we presented is not well-typed in Datafun as the interval lattice has an infinite ascending chains. To ensure termination in functional IncA, we had to introduce widening to break the infinite ascending chains of the interval lattice. Many interesting static analyses use infinite lattices with infinite ascending chains. Hence, Datafun cannot be used to express such analyses. While it is an interesting question how to guarantee termination for as many programs as possible, our system is more viable to implement real-world programs. Another difference is that Datafun has its own bottom-up semantics which was recently extended to support semi-naïve evaluation [5]. In contrast, we translate programs to Datalog and utilize off-the-shelve Datalog solvers, which readily implement semi-naïve evaluation and other optimizations.

QL [7] is a logic programming language with object-oriented features such as classes and methods to structure logic programs. QL compiles to Datalog to encode inheritance and virtual dispatch of member predicates. We also propose to compile to Datalog, but focus on functional programming with algebraic data types. It would be interesting to see how we can extend functional IncA with object-oriented features and how these interact.

Mercury [21] is a logic programming language that consists of relations and rules deriving those relations. Like any Datalog, Mercury also supports the encoding of functions as relations, but in Mercury users can additionally annotate parameters as inputs and deterministic

outputs. Mercury implements a custom Datalog solver that exploits such functional relations by executing them like a deterministic program. It would be interesting to explore *generic* Datalog optimizations that exploit functional relations, since we can easily generate the necessary annotations in functional IncA.

Bloom [2, 3] is a domain-specific language for distributed systems that uses the Datalog variant Dedalus [4] under the hood. Bloom provides built-in higher-order functions such as `map` and `reduce` that operate over collections. Bloom is embedded in Ruby and user-defined functions and data types can be written in Ruby, but these user-defined functions cannot access the contents of relations. Therefore, we cannot describe an interval analysis in the same style we have shown in the previous section in Bloom.

Soufflé [20] an efficient Datalog solver that can interpret Datalog rules directly or translate them to C++. It is possible to define user-defined functions as C++ functions, but again these functions cannot access the contents of relations. Soufflé has support for algebraic data types, but developers have to ensure that only finitely many values are constructed. For our use cases, this amounts to encoding the input relations by hand.

10 Conclusion

Datalog is supposedly declarative, but many programs are hard to express as constraints. We propose functional programming with sets as a new frontend for Datalog that solves this problem: *functional IncA*. Specifically, we translate functional IncA programs to Datalog and employ a demand transformation to ensure the Datalog program terminates whenever the original program terminates. While users of functional IncA only need to learn a single functional programming language, they enjoy Datalog's fixpoint semantics across functions and relations. Moreover, since all generated code is pure Datalog, we can use off-the-shelve Datalog solvers rather than building our own. Specifically, we implemented our approach as a frontend for IncA [23] as well as Soufflé [20] and demonstrated how easy it is to express complex Datalog programs with it. Our case studies include clone detection of real-world Java programs, program analyses, a program transformation, and an interpreter, all of which are easy to express functionally but translate to highly complex Datalog code. We have shown through early performance measurements that it is indeed desirable to use established Datalog solvers than implement custom solvers that embed a functional programming language as Formulog did. In future work, we want to investigate the performance of the generated Datalog code and study how compiler optimization can help. We also want to support negation in functional IncA, but the demand transformation potentially breaks the stratifiability of programs. We want to explore if the solution by Tekle and Liu [26] can be used. At last, we want to investigate how to properly debug functional IncA programs.

References

1 Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In Chen Li, editor, *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, pages 358–367. ACM, 2005. doi:10.1145/1065167.1065214.

2 Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In Christine Morin and Gilles Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 223–236. ACM, 2010. doi:10.1145/1755913.1755937.

3. Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260. www.cidrdb.org, 2011. URL: `http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf`.
4. Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2010. `doi:10.1007/978-3-642-24206-9_16`.
5. Michael Arntzenius and Neel Krishnaswami. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.*, 4(POPL):22:1–22:28, 2020. `doi:10.1145/3371090`.
6. Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional Datalog. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 214–227. ACM, 2016. `doi:10.1145/2951913.2951948`.
7. Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.ECOOP.2016.2`.
8. Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10(3):255–299, 1991. Special Issue: Database Logic Progamming. `doi:10.1016/0743-1066(91)90038-Q`.
9. Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.*, 4(OOPSLA):141:1–141:31, 2020. `doi:10.1145/3428209`.
10. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 243–262. ACM, 2009. `doi:10.1145/1640089.1640108`.
11. Sebastian Erdweg, Tamás Szabó, and André Pacak and. Concise, type-safe, and efficient structural diffing. In *Programming Language Design and Implementation (PLDI)*. ACM, 2021.
12. Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1213–1216. ACM, 2011. `doi:10.1145/1989323.1989456`.
13. Magnus Madsen and Ondrej Lhoták. Fixpoints for the masses: programming with first-class Datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA):125:1–125:28, 2020. `doi:10.1145/3428193`.
14. Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 194–208. ACM, 2016. `doi:10.1145/2908080.2908096`.
15. David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. ACM / Morgan & Claypool, 2018. `doi:10.1145/3191315.3191317`.

16 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999.

17 André Pacak, Sebastian Erdweg, and Tamás Szabó. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.*, 4(OOPSLA):127:1–127:28, 2020. `doi:10.1145/3428195`.

18 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. `doi:10.1023/A:1010027404223`.

19 Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. On fast large-scale program analysis in Datalog. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 196–206. ACM, 2016. `doi:10.1145/2892208.2892226`.

20 Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. A Datalog source-to-source translator for static program analysis: An experience report. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*, pages 28–37. IEEE Computer Society, 2015. `doi:10.1109/ASWEC.2015.15`.

21 Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *J. Log. Program.*, 29(1-3):17–64, 1996. `doi:10.1016/S0743-1066(96)00068-4`.

22 Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA):139:1–139:29, 2018. `doi:10.1145/3276509`.

23 Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental whole-program analysis in Datalog with lattices. In *Programming Language Design and Implementation (PLDI).* ACM, 2021.

24 Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: a DSL for the definition of incremental program analyses. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 320–331. ACM, 2016. `doi:10.1145/2970276.2970298`.

25 K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient datalog queries. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 35–44. ACM, 2010. `doi:10.1145/1836089.1836094`.

26 K. Tuncay Tekle and Yanhong A. Liu. Extended magic for negation: Efficient demand-driven evaluation of stratified Datalog with precise complexity guarantees. In Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang, editors, *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019*, volume 306 of *EPTCS*, pages 241–254, 2019. `doi:10.4204/EPTCS.306.28`.

27 Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In Avi Silberschatz, editor, *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, pages 140–149. ACM Press, 1989. `doi:10.1145/73721.73736`.

28 Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

29 Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling*, 15(3):609–629, July 2016. `doi:10.1007/s10270-016-0530-4`.

Design-By-Contract for *Flexible* Multiparty Session Protocols

Lorenzo Gheri
Imperial College London, UK

Ivan Lanese
Focus Team, University of Bologna, Italy
Focus Team, INRIA, Sophia Antipolis, France

Neil Sayers
Imperial College London, UK
Coveo Solutions Inc., Canada

Emilio Tuosto
Gran Sasso Science Institute, L'Aquila, Italy

Nobuko Yoshida
Imperial College London, UK

Abstract

Choreographic models support a correctness-by-construction principle in distributed programming. Also, they enable the automatic generation of correct message-based communication patterns from a global specification of the desired system behaviour. In this paper we extend the theory of choreography automata, a choreographic model based on finite-state automata, with two key features. First, we allow participants to act only in some of the scenarios described by the choreography automaton. While this seems natural, many choreographic approaches in the literature, and choreography automata in particular, forbid this behaviour. Second, we equip communications with assertions constraining the values that can be communicated, enabling a design-by-contract approach. We provide a toolchain allowing to exploit the theory above to generate APIs for TypeScript web programming. Programs communicating via the generated APIs follow, by construction, the prescribed communication pattern and are free from communication errors such as deadlocks.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Formal software verification

Keywords and phrases Choreography automata, design by contract, deadlock freedom, Communicating Finite State Machines, TypeScript programming

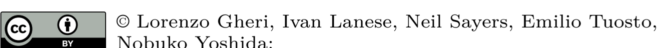
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.8

Related Version *Full Version*: `http://mrg.doc.ic.ac.uk/publications/design-by-contract-for-flexible-multiparty-session-protocols/`

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.21`
Software (Source Code): `https://github.com/Tooni/CAScript-Artifact`

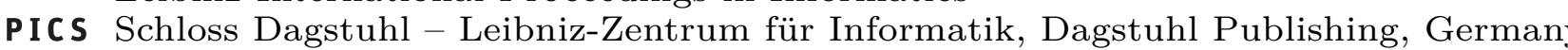
Funding Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233. Work partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems). Lanese and Tuosto are partially supported by INdAM as members of GNCS (Gruppo Nazionale per il Calcolo Scientifico). The work is partially supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T014709/1 and EP/V000462/1, and NCSS/EPSRC VeTSS.

Acknowledgements We thank the anonymous reviewers for their useful comments and suggestions. We thank Franco Barbanera for contributing to this work in its early stages. We thank Fangyi Zhou for their help with building our artifact on top of their software, νScr.



© Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 8; pp. 8:1–8:28



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The development of communicating systems is notoriously a challenging endeavour. In this application domain, both researchers and practitioners consider choreographies a valid approach to tackle software development (e.g. [28, 40, 1, 4, 15]). Besides being naturally geared toward scalability (due to the lack of central components), choreographic models have been specifically conceived to support a *correctness-by-construction* [28] principle hinging on the interplay between *global* and *local views*. The former is a description of the interactions among (the *role* of) participants. We illustrate this through an OnLineWallet (OLW) service, adapted from [39] and akin to PayPal, used by vendors to process from customers. (See [16] for a visual model of OLW). This protocol involves three participants: customer, wallet, and vendor. The former tries first to login into its account on the wallet server. In case of failure, wallet may ask for a retry, or may decide to deny access. A successful authentication is communicated by wallet to customer and vendor through the loginOK message; the vendor then sends a request for payment to customer, who can authorise or reject the transaction.

A natural question to ask is "can the OLW protocol be faithfully realised by distributed components?" The answer to this question requires a careful formalisation which we carry out in the next sections. For the moment, we appeal to intuition and interpret *realisation* as the existence of a set of components that coordinate with each other exclusively by message-passing and *faithful* as the fact that components execute all and only the communications prescribed by the global view without incurring in communication errors such as deadlocks. Local views specify the behaviour of each participant "in isolation". For instance, the local behaviour of vendor in the OLW protocol is to wait the notification from wallet, send a request message to customer, and then wait for either a payment or a rejection message from customer. Note that vendor is "oblivious" of the interactions between customer and wallet. Also, observe that, if customer fails to authenticate to wallet (e.g., by typing a wrong password), then no payment request can be made. In this case, it does not make sense to involve vendor in the protocol. We call the ability to involve a participant only in some branches of a protocol *selective participation*.

Rather than an exception, selective participation is a norm in distributed applications, e.g., for data validation, prevention of server overload, or access control. Consider, e.g., services giving public access to some resources while requiring authentication to grant access to others. Often, the authentication phase is outsourced to external services (e.g., providing OAuth2.0 [18] and Kerberos [29] authentication). In this case, accesses to public resources should be oblivious to authentication services while protected resources are not involved in the communication until the authentication phase is cleared (as for vendor in our example). Other examples of selective participation emerge from smart contracts for online money transations (e.g., crowdfunding services as [30]), where participants take part to some stages of the communication only in case of a positive outcome of some financial operation.

A paramount element for the correctness-by-construction principle is the notion of *well-formedness*, namely sufficient conditions guaranteeing the faithful realisation of a protocol. Actually, choreographies advocate the algorithmic derivation, by *projection*, of faithful realisations from well-formed global views [28]. In fact, the so-called *top-down* choreographic approach to development consists of (a) the definition of a well-formed global view of the protocol, (b) the projection of the global view onto local ones, (c) the verification that each implemented component complies with a local view.

Usually, global views abstract away from local computations; for instance, our description of OWL does not specify how wallet takes the decision of letting customer retry the authentication or the strategy of customer to authorise or not the payment. Both these (and

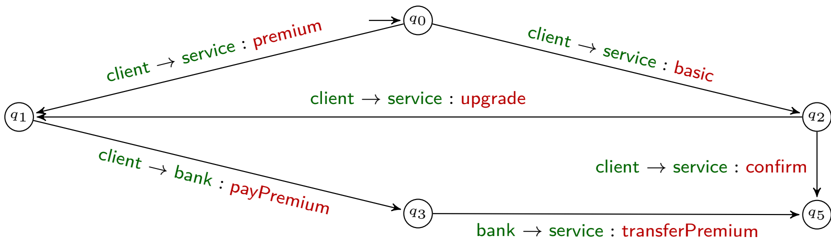


Figure 1 Non-well-structured choreography.

the other local) computations are blurred away because they require to specify the data dependencies that local computations should enforce. As pioneered in [3] in the context of global types [23], assertion methods can abstractly handle those dependencies by suitably constraining the payloads of interactions. Roughly, this transfers design-by-contract [35] methods to message-passing applications by imposing rely-guarantee relations on interactions. As shown in [3], this poses several challenges due to two main reasons. Firstly, pre-conditions ensuring the feasibility of some interactions depend on information scattered across distributed participants. Hence, it is necessary that data flow to participants so that all the information necessary for a participant to guarantee some assertion is available when needed. This requires to restrict to *history sensitive* [3] protocols, namely specifications have to be such that participants required to guarantee an assertion are aware of the information needed to satisfy it. Secondly, a careless use of such assertions may lead to inconsistent specifications so to eventually spoil the realisability of the protocol. This requires to restrict to *temporally satisfiable* [3] protocols, where no assertion ever becomes inconsistent during the execution.

Models and results based on the top-down approach to choreography abound in the literature (see, e.g., the survey [26]). This paper builds on *choreography automata* (c-automata) [2]; intuitively, a c-automaton is a finite-state machine whose transitions are labelled by interactions. The use of automata brings several benefits. On the one hand, automata models are well-known to both academics and industrial computer scientists and engineers. On the other hand, they allow one to exploit the well-developed theory of automata. Furthermore, automata do not have syntactic constraints imposed by algebraic models such as multiparty session types (see, e.g., [22, 44, 7]). Indeed, as noted in [2], c-automata seem to be more flexible than "syntax"-based formalisms such as global graphs [46] or multiparty session types. This is due to the fact that, in the latter family, well-formedness is attained via syntactic restrictions that rule out unrealisable protocols. Indeed, a distinguished feature of c-automata is that they admit *non-well-structured* interactions. Let us explain this with the c-automaton in Fig. 1, modelling a choreography where a client registers to a service according to two options. If client opts for the basic level, then no payment is due, while the premium option requires a bank payment. Thus, we have a choice at q_0 between the basic and premium service levels. Then, in state q_2 of Fig. 1, client either confirms the choice or decides to upgrade. (Selective participation is required since the bank only acts in the "left" run.) In a structured model, the "left" and the "right" runs from q_0 to q_5 must be different branches of a choice. But those models cannot encode the upgrade transition that intuitively allows one to move from one branch to the other, before the end of the choice construct.

Contribution and structure. We provide two main contributions to the theory of c-automata, as well as an implementation in the setting of TypeScript programming.

First, we extend c-automata with selective participation, which, although natural as seen above, is actually forbidden in many choreographic models (e.g., [22, 44, 7]) including c-automata [2]. For instance, we will use the OLW protocol, where vendor's involvement occurs only on successful authentication, as our running example.

Our second contribution is the definition of *asserted c-automata*, that is a design-by-contract framework for c-automata. More precisely, we equip transitions with assertions constraining the exchanged messages, allowing one to specify such policies. For example, we can specify that the authentication of OLW customer can fail at most 3 times. At a glance, asserted c-automata mimick the constructions introduced in [3]. However, the generalisation of c-automata to selective participation (not featured in [3]) and the greater flexibility introduced by non well-structured interactions require to address non-trivial technical challenges that we discuss in § 4.

The last contribution is a toolchain, dubbed CAScr, based on the theory of c-automata with selective participation developed in this paper. More precisely, CAScr allows one to specify a protocol using the Scribble framework [21, 38, 48] and to check its well-formedness relying on our theory. Finally, CAScr generates TypeScript APIs to implement the roles of the original protocol. To the best of our knowledge, CAScr is the first toolchain that integrates Scribble with the flexibility of the theory of c-automata.

Our paper is structured as follows. § 2 introduces notions on finite state automata, and in particular on communicating finite state machines, to model participants, and on c-automata.

§ 3 develops the theory of c-automata. The main novelty w.r.t. [2] is to allow for selective participation. The resulting framework is more flexible with respect to [2], e.g., it allows one to prove that the OLW protocol can be faithfully projected. Even in this more general setting we can prove standard results: the implementation has the same behaviour as the original specification (Corollary 3.13) and is free from deadlocks (Thm. 3.16). Also, when focusing on one of the participants the projected system is lock free (Thm. 3.20).

§ 4 develops our second contribution, namely design-by-contract in the setting of c-automata. More precisely, c-automata are extended with assertions (Def. 4.6) and the related theory is extended accordingly. Also in this setting the implemented system faithfully executes its specification (Corollary 4.18) and it is deadlock free (Thm. 4.19).

§ 5 presents CAScr, a novel, full toolchain – from the Scribble [21, 38, 48] specification of the communication protocol, to the generation of APIs – providing support for distributed web development in TypeScript and relying on flexible c-automata with selective participation.

Finally, § 6 discusses related work, while § 7 draws some conclusions, and sketches future directions. Proofs and auxiliary material can be found on the full version of the paper [16].

2 Choreography Automata and Communicating Systems

This section recalls basic notions about automata in general and about choreography automata (c-automata) [2] and systems of Communicating Finite State Machines (CFSMs) [5] in particular. Following [2], global views, rendered as c-automata, are projected into systems of local descriptions modelled as CFSMs. We start by surveying finite-state automata (FSA).

▶ **Definition 2.1** (FSA). *A* labelled transition system *(LTS) is a tuple* $(Q, q_0, \mathcal{L}, \mathcal{T})$ *where*

- Q *is a set of states (ranged over by* $s, q, \ldots$*) and* $q_0 \in Q$ *is the* initial state*;*
- $\mathcal{L}$ *is a finite set of labels (ranged over by* $\ell, \ldots$*);*
- $\mathcal{T} \subseteq Q \times (\mathcal{L} \cup \{\varepsilon\}) \times Q$ *is a set of transitions where* $\varepsilon \notin \mathcal{L}$ *is a distinguished label.*

A finite-state automaton *(FSA) is an LTS whose set of states is finite.*

When the LTS $A = (Q, q_0, \mathcal{L}, \mathcal{T})$ is understood we use the usual notations $s_1 \xrightarrow{\ell} s_2$ for the transition $(s_1, \ell, s_2) \in \mathcal{T}$ and $s_1 \longrightarrow s_2$ when there exists ℓ such that $s_1 \xrightarrow{\ell} s_2$, as well as $\longrightarrow^\star$ for the reflexive and transitive closure of $\longrightarrow$. We denote as $out(\mathsf{CA}, q)$ the set of transitions from q in A. We occasionally write $q \in A$ and $(q, \alpha, q') \in A$ instead of, respectively, $q \in Q$ and $(q, \alpha, q') \in \mathcal{T}$, and likewise for $_ \subseteq _$. We recall standard notions on LTSs.

▶ **Definition 2.2** (Traces and trace equivalence). *A* run *of an LTS $A = \langle \mathbb{S}, s_0, \mathcal{L}, \mathcal{T} \rangle$ is a (possibly empty) finite or infinite sequence $\pi = (s_i \xrightarrow{\ell_i} s_{i+1})_{0 \le i \le n}$ of consecutive transitions starting at s_0 (assume $n = \infty$ if the run is infinite). The* trace *(or* word*) of π is the concatenation of the labels $\mathsf{trace}(\pi)$ of the run π, namely $\mathsf{trace}(\pi) = \ell_0 \cdot \ell_1 \cdots \ell_n$. As usual, ε denotes the identity element of concatenation and the trace of an empty run is ε. Function $\mathsf{trace}(\cdot)$ extends homomorphically to sets of runs. Also, s-runs and s-traces of A are, respectively, runs and traces of $\langle \mathbb{S}, s, \mathcal{L}, \mathcal{T} \rangle$. The* language of A *is $\mathsf{L}(A) = \{\mathsf{trace}(\pi) \mid \pi$ is a run of $A\}$; A* accepts w *if $w \in \mathsf{L}(A)$ and A* accepts w from s *if $w \in \mathsf{L}(\langle \mathbb{S}, s, \mathcal{L}, \mathcal{T} \rangle)$. LTSs A and B are* trace equivalent *if $\mathsf{L}(A) = \mathsf{L}(B)$.*

Bisimilarity [42] is an equivalence relation on LTSs simpler to prove than trace equivalence which is implied by bisimilarity, and coincides with it for deterministic LTSs.

▶ **Definition 2.3** (Bisimulation). *Let $A = \langle \mathbb{S}_A, s_{0A}, \mathcal{L}, \mathcal{T}_A \rangle$ and $B = \langle \mathbb{S}_B, s_{0B}, \mathcal{L}, \mathcal{T}_B \rangle$ be two LTSs. A relation $\mathcal{R} \subseteq (\mathbb{S}_A \times \mathbb{S}_B) \cup (\mathbb{S}_B \times \mathbb{S}_A)$ is a* (strong) bisimulation *if it is symmetric, $(s_{0A}, s_{0B}) \in \mathcal{R}$, and for every pair of states $(p, q) \in \mathcal{R}$ and all labels ℓ:*

if $p \xrightarrow{\ell} p'$ then there is $q \xrightarrow{\ell} q'$ such that $(p', q') \in \mathcal{R}$

Relation $\mathcal{R}$ is a weak *bisimulation if it is symmetric, $(s_{0A}, s_{0B}) \in \mathcal{R}$, and for every pair of states $(p, q) \in \mathcal{R}$ and all labels ℓ:*

- *if $p \xrightarrow{\ell} p'$ with $\ell \neq \varepsilon$ then there is a run $q \xrightarrow{\varepsilon}^\star \xrightarrow{\ell} \xrightarrow{\varepsilon}^\star q'$ such that $(p', q') \in \mathcal{R}$ and*
- *if $p \xrightarrow{\varepsilon} p'$ then there is a run $q \xrightarrow{\varepsilon}^\star q'$ such that $(p', q') \in \mathcal{R}$.*

If two LTSs are bisimilar then they are also trace equivalent.

A main role in our models is played by *interactions* built on the alphabet:

$$\mathcal{L}_{\text{int}} \triangleq \{ \mathsf{p} \to \mathsf{q} : \mathsf{m} \mid \mathsf{p} \neq \mathsf{q} \in \mathfrak{P} \quad \text{and} \quad \mathsf{m} \in \mathcal{M} \} \text{ ranged over by lowercase Greek letters}$$

where $\mathfrak{P}$ and $\mathcal{M}$ are, respectively, sets of participants and of messages. We assume $\mathfrak{P} \cap \mathcal{M} = \emptyset$. An interaction $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ specifies that participant p sends a message (of type) m to participant q and participant q receives m. Hence, by construction, each send is paired with a unique receive and vice versa. In most choreographic models, this forbids to specify message losses, races, and deadlocks. Adopting the terminology of the session type community (see, e.g., [26]),

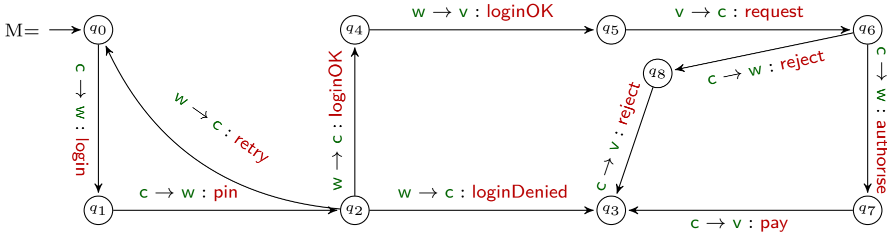
- with *message loss* we mean a send that cannot be matched by a receive; this cannot happen in interactions since $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ specifies both the send and the receive together;
- with *race* we mean a configuration where a receiver non-deterministically interacts with either of two senders (or a sender with either of two receivers), depending on the relative speed of their execution; this cannot happen since an interaction specifies which send is supposed to interact with which receive and vice versa (notably, concurrency can take place without message races, e.g., if participant p sends to participant q and at the same time c sends to d there is no race);
- with *deadlock* we mean a configuration where two or more participants are blocked waiting for one another forming cyclic dependencies (e.g., p is waiting for q which waits for c, which in turns waits for p); this cannot happen either since an interaction specifies which participant has to send and which one has to receive.

All these properties hold by construction in most choreographic models. However, care is needed to ensure that these properties are preserved when moving from the choreographic specification to a distributed implementation. Such analysis has been performed for many choreographic models in the literature (see [26]).

▶ **Definition 2.4** (Choreography automata). *A* choreography automaton *(c-automaton) is an FSA on the alphabet $\mathcal{L}_{int}$. Elements of $\mathcal{L}_{int}^{\star} \cup \mathcal{L}_{int}^{\omega}$ are* choreography words*, subsets of $\mathcal{L}_{int}^{\star} \cup \mathcal{L}_{int}^{\omega}$ are* choreography languages*.*

The set of participants of a c-automaton is finite; we denote with $\mathcal{P}_{\mathsf{CA}}$ (or simply $\mathcal{P}$ if CA is understood) the set of participants of c-automaton CA. Given $\mathsf{p} \to \mathsf{q} : \mathsf{m} \in \mathcal{L}_{\text{int}}$, we define $\mathsf{ptp}(\mathsf{p} \to \mathsf{q} : \mathsf{m}) \triangleq \{\mathsf{p}, \mathsf{q}\}$ and extend it homorphically to (sets of) transitions. We say that $\alpha, \beta \in \mathcal{L}_{\text{int}}$ are *independent*, written $\alpha \parallel \beta$, if $\mathsf{ptp}(\alpha) \cap \mathsf{ptp}(\beta) = \emptyset$.

▶ **Example 2.5** (OLW's c-automaton). The c-automaton



models the OLW example in § 1. ⌟

We now survey communicating systems [5], our formal model of local views.

▶ **Definition 2.6** (Communicating system). *A* communicating finite-state machine *(CFSM) is an FSA on the set $\mathcal{L}_{act} \triangleq \{\mathsf{pq!m}, \mathsf{pq?m} \mid \mathsf{p}, \mathsf{q} \in \mathfrak{P} \text{ and } \mathsf{m} \in \mathcal{M}\}$ of* actions.

Action $\mathsf{pq!m}$ is the send of message m from p to q, while action $\mathsf{pq?m}$ is the corresponding receive. The subjects *of an output and an input action, say $\mathsf{pq!m}$ and $\mathsf{pq?m}$, are respectively p and q. A CFSM is* p-local *if all its transitions have labels with subject p. A* (communicating) system *is a map $\mathsf{S} = (M_{\mathsf{p}})_{\mathsf{p} \in \mathcal{P}}$ assigning a p-local CFSM M_{p} to each participant $\mathsf{p} \in \mathcal{P}$. We require that $\mathcal{P} \subseteq \mathfrak{P}$ is finite and that any participant occurring in a transition of M_{p} is in $\mathcal{P}$.*

We now introduce the notion of projection from c-automata to systems of CFSMs. Intuitively, projection builds a system aimed at implementing the projected c-automaton. Similar notions in the literature often take the name of endpoint projection (see, e.g., [23, 7]).

▶ **Definition 2.7** (Automata projection). *The* projection $\alpha\!\downarrow_{\mathsf{p}}$ of an interaction α on $\mathsf{p} \in \mathfrak{P}$ *is*

$$(\mathsf{p} \to \mathsf{q} : \mathsf{m})\!\downarrow_{\mathsf{p}} = \mathsf{pq!m}, \quad (\mathsf{q} \to \mathsf{p} : \mathsf{m})\!\downarrow_{\mathsf{p}} = \mathsf{qp?m}, \qquad \textit{and} \qquad \alpha\!\downarrow_{\mathsf{p}} = \varepsilon \textit{ for any other label } \alpha$$

Function $_\!\downarrow_{\mathsf{p}}$ extends homomorphically to transitions, runs, and choreography words.

The projection $\mathsf{CA}\!\downarrow_{\mathsf{p}}$ *of a c-automaton $\mathsf{CA} = \langle \mathbb{S}, q_0, \mathcal{L}_{int}, \mathcal{T} \rangle$ on a participant $\mathsf{p} \in \mathcal{P}$ is obtained by determinising and minimising up-to language equivalence the* intermediate *CFSM*

$$A_{\mathsf{p}} = \left\langle \mathbb{S}, q_0, \mathcal{L}_{act}, \left\{ (q \xrightarrow{\alpha\downarrow_{\mathsf{p}}} q') \mid q \xrightarrow{\alpha} q' \in \mathcal{T} \right\} \right\rangle$$

The projection of CA*, written $\mathsf{CA}\!\downarrow$, is the communicating system $(\mathsf{CA}\!\downarrow_{\mathsf{p}})_{\mathsf{p} \in \mathcal{P}}$.*

▶ **Example 2.8** (Projecting OLW). We instantiate here projection on the c-automaton for the OLW protocol described in Ex. 2.5. In particular, the intermediate CFSM A_{v} is

the determinisation of which yields the following CFSM $\mathsf{CA}{\downarrow_{\mathsf{v}}}$ for the vendor participant

Noteworthy, due to determinisation, states of the projection correspond to (not necessarily disjoint) sets of states of the starting c-automaton. Indeed, in $\mathsf{CA}\downarrow_{\mathsf{v}}$ we have $Q_0 = \{q_0, q_1, q_2, q_3, q_4\}$, $Q_1 = \{q_5\}$, $Q_2 = \{q_6, q_7, q_8\}$, and $Q_2 = \{q_3\}$. ⌟

We present below the semantics of communicating systems. We consider a synchronous semantics. Essentially, a system can execute an interaction $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ if two of its participants can provide complementary actions $\mathsf{pq!m}$ and $\mathsf{pq?m}$ (while the others do not move), and can take an ϵ action if one of its participant can do it (while the others do not move).

▶ **Definition 2.9** (Semantics of communicating systems). *Let $\mathsf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ be a communicating system where $M_{\mathsf{p}} = \langle \mathbb{S}_{\mathsf{p}}, q_{0\mathsf{p}}, \mathcal{L}_{act}, \mathcal{T}_{\mathsf{p}}\rangle$ for each participant $\mathsf{p} \in \mathcal{P}$.*

A configuration *of S is a map $s = (q_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ assigning a* local state *$q_{\mathsf{p}} \in \mathbb{S}_{\mathsf{p}}$ to each $\mathsf{p} \in \mathcal{P}$. The* semantics *of S is the c-automaton $[\![\mathsf{S}]\!] = \langle \mathbb{S}, s_0, \mathcal{L}_{int}, \mathcal{T}\rangle$ where*

- *$\mathbb{S}$ is the set of configurations of S, as defined above, and $s_0 : \mathsf{p} \mapsto q_{0\mathsf{p}}$ for each $\mathsf{p} \in \mathcal{P}$ is the* initial *configuration of $\mathbb{S}$*
- *$\mathcal{T}$ is the set of transitions*
  - *$s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} s_2$ such that*
    - *$s_1(\mathsf{p}) \xrightarrow{\mathsf{pq!m}} s_2(\mathsf{p}) \in \mathcal{T}_{\mathsf{p}}$ and $s_1(\mathsf{q}) \xrightarrow{\mathsf{pq?m}} s_2(\mathsf{q}) \in \mathcal{T}_{\mathsf{q}}$, and*
    - *for all $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p}, \mathsf{q}\}$, $s_1(\mathsf{x}) = s_2(\mathsf{x})$*
  - *$s_1 \xrightarrow{\varepsilon} s_2$ such that $s_1(\mathsf{p}) \xrightarrow{\varepsilon} s_2(\mathsf{p}) \in \mathcal{T}_{\mathsf{p}}$, and for all $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p}\}$, $s_1(\mathsf{x}) = s_2(\mathsf{x})$.*

3 Flexible Choreography Automata

We now introduce a theory of c-automata enabling faithful realisations, which is formalised as language equivalence between a c-automaton and the semantics of its projection as proved in Corollary 3.13. However, not all c-automata can be faithfully realised, hence we need to restrict to *well-formed* c-automata. Well-formedness is defined as the conjunction of two properties, *well-sequencedness* and *well-branchedness*. Both these properties are inspired from [2]. However, well-branchedness is generalised to allow participants to act on some of the scenarios specified by the c-automaton only upon request from other participants. We call this feature *selective participation*, since a participant may act on a branch only if selected to be involved by some other participant. This is disallowed in many choreographic formalisms (e.g., [22, 44, 7]), including choreography automata [2]. On the other hand, well-sequencedness is strengthened since the formulation in [2] is not enough to ensure faithful realisations. We start by defining concurrent transitions, exploited in the definition of well-sequencedness.

▶ **Definition 3.1** (Concurrent transitions). *Two consecutive transitions* $q \xrightarrow{\alpha} q' \xrightarrow{\beta} q''$ *are* concurrent *if there is* q''' *such that* $q \xrightarrow{\beta} q''' \xrightarrow{\alpha} q''$.

Essentially, two transitions are concurrent if they give rise to a commuting diamond.

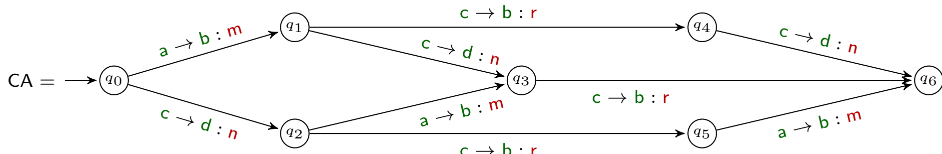
▶ **Definition 3.2** (Well-sequencedness). *A c-automaton* CA *is* well-sequenced *if for each two consecutive transitions* $q \xrightarrow{\alpha} q' \xrightarrow{\beta} q''$ *either*

(a) $\alpha \not\parallel \beta$*, i.e.,* α *and* β *are not independent (hence* $ptp(\alpha) \cap ptp(\beta) \neq \emptyset$*), or*

(b) *there is* q''' *such that* $q \xrightarrow{\beta} q''' \xrightarrow{\alpha} q''$ *(i.e., the transitions are concurrent); furthermore for each transition* $q''' \xrightarrow{\gamma} q''''$*,* $\gamma \parallel \alpha$ *and* $\gamma \parallel \beta$*.*

Intuitively, well-sequencedness forces the explicit representation of concurrency among interactions with disjoint sets of participants as commuting diamonds. The second part of clause (b) in Def. 3.2 rules out the entanglement of choices with commuting diamonds, while enabling to compose an arbitrary number of independent actions. This condition, absent in [2], does not allow them to enforce faithful realisations as shown in the next example.

▶ **Example 3.3.** Consider the c-automaton below.



In CA↓, participant c can immediately send r to b, since it is not involved in transition $q_0 \xrightarrow{\mathsf{a \to b:m}} q_1$. Similarly, b can immediately receive r from c, since it is not involved in transition $q_0 \xrightarrow{\mathsf{c \to d:n}} q_2$. Thus, a transition with label $\mathsf{c \to b : r}$ is enabled in the initial configuration of the semantics of CA↓. However, no transition with the same label is enabled in the initial state of CA, hence the implementation is not faithful. ⌟

The following auxiliary concepts are instrumental in the definition of well-branchedness (cf. Def. 3.7). Given a word w, $pref(w)$ denotes the set of its prefixes.

▶ **Definition 3.4** (Full awareness). *Let* (π_1, π_2) *be a pair of q-runs of a c-automaton* CA*. Participant* $\mathsf{p} \in ptp(\pi_1) \cap ptp(\pi_2)$ *is* fully aware *of* (π_1, π_2) *if there are* $\alpha_1 \neq \alpha_2 \in \mathcal{L}_{int}$ *such that* $\mathsf{p} \in ptp(\alpha_1) \cap ptp(\alpha_2)$ *and*

1. *either* α_h *is the first interaction in* $L(\pi_h)$ *for* $h = 1, 2$
2. *or for* $h \in \{1, 2\}$ *there is a proper prefix* $\hat{\pi}_i$ *of* π_i *such that* $trace(\hat{\pi}_1{\downarrow_\mathsf{p}}) = trace(\hat{\pi}_2{\downarrow_\mathsf{p}})$*, the partners of* p *in* α_h *are fully aware of* $(\hat{\pi}_1, \hat{\pi}_2)$*,* $trace(\hat{\pi}_h)\alpha_h \in pref(trace(\pi_h))$*, and* α_h *does not occur on* π_{3-h}*.*

Intuitively, a participant p is fully aware of two q-runs when able to ascertain which branch has been taken. This happens either when p itself chooses (1), or when p is informed of the choice by interacting with some other participant already fully aware of the q-runs (2).

▶ **Example 3.5** (Full awareness in OLW). Let us consider the runs $\pi_1 = q_2 \xrightarrow{\mathsf{w \to c:loginOK}} q_4 \xrightarrow{\mathsf{w \to v:loginOK}} q_5$ and $\pi_2 = q_2 \xrightarrow{\mathsf{w \to c:loginDenied}} q_3$ of the OLW c-automaton M in Ex. 2.5. Both w and c are fully-aware of (π_1, π_2) since they occur in the first interaction in both the runs (Def. 3.4(1)). Participant v is not fully-aware of (π_1, π_2) since it occurs on π_1 only.

Take now the runs $\pi_3 = q_6 \xrightarrow{\mathsf{c \to w:reject}} q_8 \xrightarrow{\mathsf{c \to v:reject}} q_3$ and $\pi_4 = q_6 \xrightarrow{\mathsf{c \to w:authorise}} q_7 \xrightarrow{\mathsf{c \to v:pay}} q_3$ in M. As before, both participants w and c are fully-aware of (π_3, π_4) since they occur in the first interaction in both the runs. Participant v is fully-aware of (π_3, π_4) as well, since its partner c is fully-aware of $(q_6 \xrightarrow{\mathsf{c \to w:reject}} q_8, q_6 \xrightarrow{\mathsf{c \to w:authorise}} q_7)$. ⌟

To establish well-branchedness of a c-automaton we have to ensure that for each choice, namely for each state with (at least) two non-independent outgoing transitions, and each participant p, if p has to take different actions in the branches starting from the two transitions, then p is fully-aware of the taken branch. In principle, such a condition should be checked on all pairs of coinitial paths. However, this would lead to redundant checks, hence below we borrow from [2] the notion of q-spans, namely pairs of paths from q on which we will perform the check. Essentially, we have to handle choices with loops on some branches and we have to consider "long-enough" branches. More precisely, a q-run in a c-automaton CA is a *pre-candidate q-branch* if each of its cycles has at most one occurrence within the whole run (i.e., if π' is a q'-run included in π and ending in q', then π' has exactly one occurrence in π); a *candidate q-branch* is a maximal pre-candidate q-branch with respect to the prefix order.

▶ **Definition 3.6** (q-span)**.** *A pair (π, π') of pre-candidate q-branches of* CA *is a q-span if*
1. *either π and π' are cofinal, with no common node but q and the last one;*
2. *or π and π' are candidate q-branches with no common node but q;*
3. *or π and π' are a candidate q-branch and a loop on q with no other common nodes.*

We can now introduce well-branchedness.

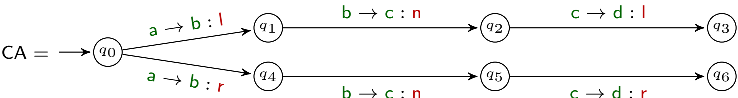
▶ **Definition 3.7** (Well-branchedness)**.** *A c-automaton* CA *is* well-branched *if it is deterministic and for each of its states q there is a partition $T_1, \dots, T_k$ of $out(\mathsf{CA}, q)$ such that*
- *for all $1 \le i \ne j \le k$, $ptp(T_i) \cap ptp(T_j) = \emptyset$ and for each $q \xrightarrow{\alpha_i} q_i \in T_i$, $q \xrightarrow{\alpha_j} q_j \in T_j$ there exists q' such that $q_i \xrightarrow{\alpha_j} q'$ and $q_j \xrightarrow{\alpha_i} q'$*
- *for all $1 \le i \le k$, $\bigcap_{t \in T_i} ptp(t) \ne \emptyset$ and for all $\mathsf{p} \in ptp(\mathsf{CA}) \setminus \bigcap_{t \in T_i} ptp(t)$ and q-span (π_1, π_2) starting from transitions in T_i, if $\pi_1 \downarrow_{\mathsf{p}} \ne \pi_2 \downarrow_{\mathsf{p}}$ then either* p *is fully aware of (π_1, π_2) or there is $i \in \{1, 2\}$ such that $\mathsf{p} \notin ptp(\pi_i)$ and*
  1. *the first transition in π_{3-i} involving* p *is with a fully aware participant of (π_1, π_2) and*
  2. *for all runs π' such that $\pi_i\pi'$ is a candidate q-branch of* CA *the first transition in π' involving* p *is with a participant which is fully aware of (π_1, π_2).*

Intuitively, a c-automaton is well-branched if for any state with multiple outgoing transitions (both clauses in Def. 3.7 trivially hold when $out(\mathsf{CA}, q)$ is empty or a singleton) , we can group them in equivalence classes. Transitions in different classes are concurrent, hence they give rise to commuting diamonds. Transitions in the same class are choices: one participant, belonging to all the (initial) transitions, makes the choice, and any other participant p is either fully aware of the q-runs or it is inactive in some branch π_i (condition $\mathsf{p} \notin ptp(\pi_i)$). In the last case, p has to interact with a fully aware partner (i) on each continuation π' (if any) of π_i as well as (ii) inside the other branch, π_{3-i}. Intuitively, (i) is necessary to make p aware of when the choice is fully completed and (ii) on whether the branch on which p needs to act has been taken. At the price of increasing the technical complexity, the second clause in Def. 3.7 can be relaxed. Indeed, right now it requires a participant p, occurring in one branch only, to interact (both in the branch where it occurs and in the continuations after the merge of the two branches) with a fully-aware participant. We could instead allow p to interact with a chain of other participants occurring only in the same branch, and such that the last participant in the chain interacts with a fully-aware participant.

▶ **Example 3.8** (OLW is well-branched)**.** Let us show that the c-automaton in Ex. 2.5 is well-branched. The only states for which well-branchedness is not trivial are q_2 and q_6 (the others have at most one outgoing transition). In both the cases we have a single equivalence class where w and c are in all the first transitions; hence they are both fully-aware in all the possible spans. Let us check the condition for v. Let us consider q_6. There is one

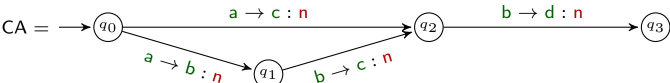
q_6-span, with branches with states q_6, q_8, q_3 and q_6, q_7, q_3, which fits case 1 in Def. 3.6. As discussed in Ex. 3.5, in this q_6-span v is fully-aware, hence the condition is satisfied. Let us now consider q_2. Here we have a loop with states q_2, q_0, q_1, q_2, a candidate q_2-branch with states q_2, q_3, and two candidate q_2-branches with a common prefix (states q_2, q_4, q_5, q_6) and two continuations (states q_6, q_8, q_3 and q_6, q_7, q_3). Any combination of the self-loop with the candidate q_2-branches fit in case 3 in Def. 3.6, while the pairings of the first candidate q_2-branch with any of the others fit in case 1 in Def. 3.6. In the q_2-spans above v occurs only in the one with two continuations. Since there it interacts with c which is fully-aware, condition 1 in Def. 3.7 holds. Condition 2 holds trivially, since the branches join only in state q_3 which has no outgoing transitions. ⌟

▶ **Example 3.9** (Non well-branched c-automata)**.** Consider the c-automaton below.



Here, c is not fully-aware since it interacts with b (which is fully-aware) receiving the same message on both the branches. Hence, its first different interactions are with d, which is not fully-aware. Indeed, d gets different messages, but from c which is not fully aware either. Thus, c and d can decide, e.g., to take the lower branch even if a and b took the upper one, thus producing a trace $a \to b : l \cdot b \to c : n \cdot c \to d : r$ not part of the language of CA. ⌟

▶ **Example 3.10** (Non well-branchedness with selective participation)**.** Consider the c-automaton:



Here, b occurs in the bottom branch only, interacting with a which is fully-aware, as required. However, after the merge of the two branches, b interacts with d which is not fully aware, thus violating condition 2 in Def. 3.7. Indeed the interaction $b \to d : n$ is enabled since the initial configuration, against the prescription of CA. ⌟

▶ **Definition 3.11** (Well-formedness)**.** *A c-automaton* CA *is* well-formed *if it is both well-sequenced and well-branched.*

Well-formed c-automata enjoy relevant properties. First, for each well-formed c-automaton the semantics of the projection is bisimilar to the starting c-automaton.

▶ **Theorem 3.12.** CA *is bisimilar to* $[\![\mathsf{CA}\!\downarrow]\!]$ *for any well-formed c-automaton* CA*.*

An immediate consequence of Thm. 3.12 is that the language of a well-formed c-automaton coincides with the language of the semantics of its projection.

▶ **Corollary 3.13.** $L(\mathsf{CA}) = L([\![\mathsf{CA}\!\downarrow]\!])$ *for any well-formed c-automaton* CA*.*

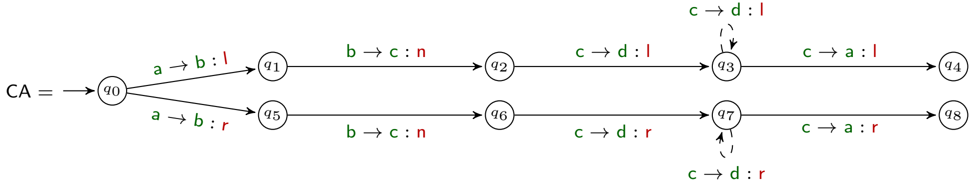
We now show that projections of well-formed c-automata do not deadlock. To this end, we need to extend CFSMs with a concept of final state. Intuitively, a state is final in the projection on some participant p of a given c-automaton CA iff one of the corresponding states of CA (remember that states of the projection are sets of states of CA) has an outgoing maximal path along with p is not involved. Formally:

▶ **Definition 3.14** (Final states in projected CFSMs). *Let* CA *be a c-automaton and* p *one of its participants. A state* Q *of* $\mathsf{CA}\!\downarrow_{\mathsf{p}}$ *is* final *if in* CA *there is* $q \in Q$ *and a candidate* q*-branch* π *such that* $\mathsf{p} \notin \mathit{ptp}(\pi)$*.*

▶ **Definition 3.15** (Deadlock freedom). *The projection of a c-automaton is* deadlock-free *if for each of its reachable configurations* s *either* s *has an outgoing transition or, for each participant* p*,* $s(\mathsf{p})$ *is final.*

▶ **Theorem 3.16** (Projections of well-formed c-automata are deadlock-free). *Let* CA *be a well-formed c-automaton. Then* $\mathsf{CA}\!\downarrow$ *is deadlock-free.*

▶ **Example 3.17** (C-automaton with deadlock). Consider the c-automaton



obtained by adding the transitions from states q_3 and q_7 to the one in Ex. 3.9. Disregard the dashed transitions. If, as discussed in Ex. 3.9, c and d decide to take the bottommost branch while a and b take the uppermost one, we can reach a configuration s where c wants to send r to a, but a is only willing to take l. Hence, no transition is possible and we have a deadlock. Due to Thm. 3.16 this is possible only since the c-automaton is not well-formed. ⌟

We can refine the result above by focusing on a single participant.

▶ **Definition 3.18** (Lock freedom). *The projection of a c-automaton is* lock-free *if for each of its reachable configurations* s *and each participant* p*, either* $s(\mathsf{p})$ *is final or* s *has at least an outgoing transition and for each candidate* s*-branch* π *we have* $\mathsf{p} \in \mathit{ptp}(\pi)$*.*

Lock freedom is strictly stronger than deadlock freedom. Indeed, each configuration s and a participant p such that $s(\mathsf{p})$ is not final has an outgoing transition, hence it is not a deadlock. However, there are systems which are deadlock-free but not lock-free, as discussed below.

▶ **Example 3.19** (C-automaton with locks (but no deadlock)). Consider again the c-automaton from Ex. 3.17, including the dashed self-loops. There is now no deadlock, since the configuration s has an outgoing transition, namely a self-loop involving c and d. However, s is a lock for a. Indeed, it is not final for a, yet a does not take part in the branch corresponding to the execution of the self-loop.

▶ **Theorem 3.20** (Projections of well-formed c-automata are lock-free). *Let* CA *be a well-formed c-automaton. Then* $\mathsf{CA}\!\downarrow$ *is lock-free.*

4 Design-by-Contract

We now extend the theory of choreography automata and communicating systems to handle specifications amenable to predicate over data exchanged through a protocol. The basic idea is to frame the design-by-contract theory proposed in [3] for global types in the context of c-automata. This theory advocates *global assertions* to specify and verify contracts among participants of a protocol. Taking inspiration from Design-by-Contract (DbC) [35], widely used in the practice of sequential programming [20, 14], a global assertion is a global type decorated with logical formulae predicating on the payload carried by interactions. Just as in the traditional DbC, the use of logical predicates allows one to specify protocols where the content of messages is somehow constrained.

4.1 Asserted choreography automata

To specify protocols that encompass constraints on payloads, we extend c-automata to *asserted c-automata*. The structure of messages is reshaped to account for sorted data in interactions and predicate over the payload of communications. More precisely, the set of *messages* $\mathcal{M}$ consists of *tagged tuples* $\tau\,\langle \mathtt{V}\rangle$ where τ is a *tag* and $\mathtt{V} = \mathsf{v}_1\ \mathsf{s}_1, \ldots, \mathsf{v}_h\ \mathsf{s}_h$ is a tuple of pairwise distinct sorted variables (namely, $\mathsf{v}_i = \mathsf{v}_j \implies i = j$ for $1 \leq i \leq j \leq h$). The set of variables of $\mathtt{V} = \mathsf{v}_1\ \mathsf{s}_1, \ldots, \mathsf{v}_h\ \mathsf{s}_h$ is $\mathsf{var}(\mathtt{V}) \triangleq \{\mathsf{v}_1, \ldots, \mathsf{v}_h\}$ and, accordingly $\mathsf{var}(\mathsf{m}) \triangleq \mathsf{var}(\mathtt{V})$ and $\mathsf{var}(\mathsf{p} \to \mathsf{q} : \mathsf{m}) \triangleq \mathsf{var}(\mathsf{m})$ are the set of variables of m and of $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ respectively. Intuitively, now an interaction specifies also the sort of the values communicated by the sender and the "local" variables where the receiver "stores" those values.

▶ **Example 4.1** (OLW variable sorts). When asking customer for another login attempt, wallet can send a message retry $\langle$msg string$\rangle$ where the payload msg yields an error message. ⌟

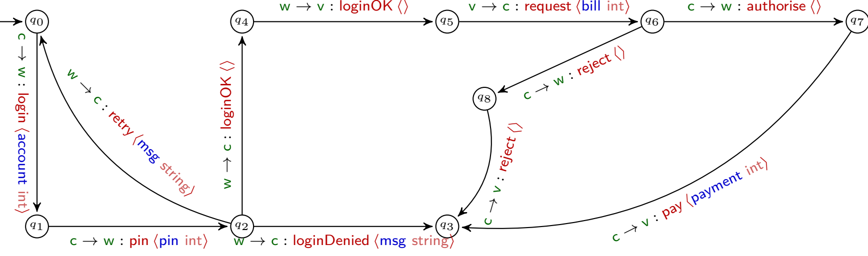
We borrow from [3] (with minor syntactic changes) the first-order logic to specify the constraints on payloads; the set $\mathcal{A}$ of logical formulae are derived from the following grammar

$$\mathsf{A}, \mathsf{B} ::= \top \mid \bot \mid \phi(e_1, \ldots, e_n) \mid \neg \mathsf{A} \mid \mathsf{A} \wedge \mathsf{B} \mid \mathsf{A} \supset \mathsf{B} \mid \exists \mathsf{v}\ \mathsf{s} :\ \mathsf{A} \qquad (1)$$

In (1), ϕ ranges over pre-defined atomic predicates with fixed arities and sorts (e.g., bool, int, etc) [34, §2.8] and $e_1, \ldots, e_n$ denote expressions. Instead of fixing a specific language of expressions, we just assume that they encompass usual data types of programming languages and variables v. Also, we assume that sorts of expressions can be inferred (hence, we occasionally omit sorts and tacitly assume that usage of variables is consistent with respect to their sort). For simplicity, we consider only basic sorts (as in [3]). More complex static data structures can be handled similarly, while dynamic data structures (e.g., pointers) require to extend our theory with suitable semantics of value passing (e.g., deep-copy).

Let $\mathsf{var}(e)$ be the set of variables occurring in expression e; likewise $\mathsf{var}(\mathsf{A})$ denotes the set of free variables of predicate $\mathsf{A} \in \mathcal{A}$, while $\mathsf{bvar}(\mathsf{A})$ denotes the bound variables in A (defined in the standard way). Hereafter, assume that $\mathsf{var}(\mathsf{A}) \cap \mathsf{bvar}(\mathsf{A}) = \emptyset$.

▶ **Example 4.2** (OLW payloads). The payloads of the OLW protocol which we will use through the paper are those in the following FSA:



Notice that some messages have empty payloads. ⌟

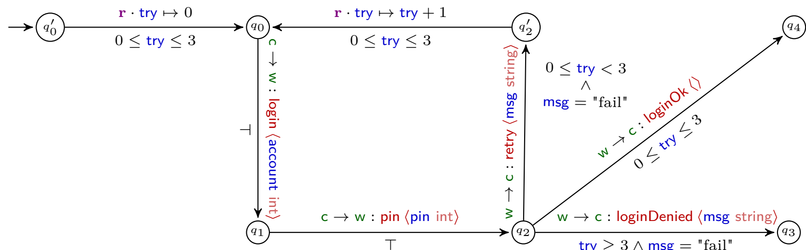
We will consider FSAs where transitions are decorated with *assertions*, namely formulae in $\mathcal{A}$ predicating on variables of the FSAs. The interplay between payloads and assertions requires some care to handle iterative behaviour and the scoping of variables. In fact, we will need to slightly change the FSA above to handle the iteration of the authentication phase.

Iterative computations require a few more ingredients. First we fix a *recursion context* ρ which maps each recursion variable $\mathbf{r}$ to a triplet $(\mathtt{V}, \mathsf{A}, q)$ consisting of

- a set of sorted variables $\mathtt{V}$ which identify the formal parameters of $\mathbf{r}$,
- a predicate $\mathsf{A} \in \mathcal{A}$, the loop invariant to be maintained through the iteration, and
- a state q of the FSA identifying the start of the iteration.

We assume that if $\rho(\mathbf{r}) = (\mathtt{V}, \mathsf{A}, q)$ and $\rho(\mathbf{r}') = (\mathtt{V}', \mathsf{A}', q')$ then $\mathbf{r} \neq \mathbf{r}'$ implies $q \neq q'$ and $\mathtt{V} \cap \mathtt{V}' = \emptyset$. Then we use FSAs on the set $\widehat{\mathcal{L}_{\text{int}}}$ (ranged over by λ), defined as the union of $\mathcal{L}_{\text{int}}$ and the set of *recursive calls* which are defined as pairs $\mathbf{r} \cdot \iota$ of a recursive variable and a map assigning expressions to recursive parameters of $\mathbf{r}$.

▶ **Example 4.3** (OLW iteration). Using assertions, the constraint on the authentication phase of the OLW protocol described in § 1 can be specified as follows:



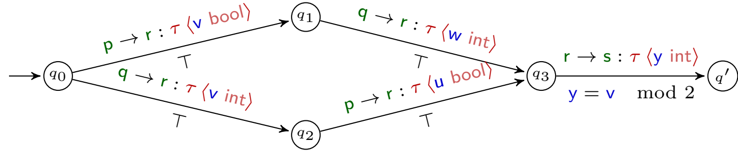
where $\rho(\mathbf{r}) = (\{\mathsf{try}\}, 0 \leq \mathsf{try} \leq 3, q_0)$. The automaton above refines the left part of the c-automaton in Ex. 4.2. In particular, states with the same names do correspond. States q_0' and q_2' are new (in particular q_0' is the new initial state), introduced to correctly model iteration. The assertions on the transitions from states q_0' and q_2' model recursive calls where the try parameter is respectively set to 0 and incremented (cf. Ex. 4.5). ⌟

Transitions $t = (q, (\lambda, \mathsf{A}), q')$, written as $q \xrightarrow[\mathsf{A}]{\lambda} q'$, are interpreted according to their label:

- If $\lambda = \mathsf{p} \to \mathsf{q} : \mathsf{m}$ then t (dubbed *interaction transition*) establishes a rely-guarantee relation: when t is fired, p *guarantees* A while q *assumes* that A holds.
- If $\lambda = \mathbf{r} \cdot \iota$ then t (dubbed *iteration transition*) records the invariant A (fixed by the recursion context ρ) that should be maintained through each loop corresponding to $\mathbf{r}$.

Variable scoping requires attention, as best illustrated by the following example.

▶ **Example 4.4** (Confusion). In the following FSA



it is not clear if the assertion on the transition from q_3 predicates on the variable v bound in the interaction between p and r or in the one between q and r, hence its sort is not clear. ⌟

The binding and scoping of variables yield a first difference w.r.t. [3], where syntactic structures of global assertions facilitate the definition of these notions. The lack of syntactic structures of c-automata requires instead to introduce constructions to handle variables.

Let us now consider recursion. An FSA A *respects* a recursion context ρ when there are no loops without iteration transitions and for each iteration transition $t = q \xrightarrow[\mathsf{A}]{\mathbf{r} \cdot \iota} \hat{q}$ in A with $\rho(\mathbf{r}) = (\mathtt{V}, \mathsf{A}, \hat{q})$

(a) t is the only outgoing transition of q and $q \neq \hat{q}$ and
(b) either q is the initial state of A or there is a unique transition entering q and it is an interaction transition.

Condition (a) forbids self-loops while (b) forces iterations to be guarded by interactions.

▶ **Example 4.5** (OLW is respectful)**.** The requirements imposed by respectfulness are met by the FSA in Ex. 4.3. ⌟

For an FSA $A = (Q, q_0, \widehat{\mathcal{L}_{\text{int}}} \times \mathcal{A}, \mathcal{T})$ on $\widehat{\mathcal{L}_{\text{int}}} \times \mathcal{A}$, we let $\mathsf{SPath}_A(q)$ denote the set of simple paths[1] reaching the state $q \in Q$ from q_0; also, $\mathsf{var}(q \xrightarrow[\mathsf{A}]{\alpha} q') \triangleq \mathsf{var}(\alpha)$ and $\mathsf{var}(q \xrightarrow[\mathsf{A}]{\mathbf{r} \cdot \iota} q') \triangleq \mathsf{var}(\mathbf{r}) \triangleq \mathtt{V}$ if $\rho(\mathbf{r}) = (\mathtt{V}, \mathsf{A}, \hat{q})$. Finally, we say that a transition $t \in \mathcal{T}$ from a state $q \in Q$ *fixes a variable* v *(in A)* if $\mathsf{v} \in \mathsf{var}(t)$ and, for each path $\pi \in \mathsf{SPath}_A(q)$ there is no transition $t' \in \pi$ that fixes v.

The next definition addresses the issues of confusion and respectfulness described above.

▶ **Definition 4.6** (Asserted c-automata)**.** *An FSA, say* CA*, on the alphabet* $\widehat{\mathcal{L}_{int}} \times \mathcal{A}$ *such that*
1. *for each co-final span* (π, π') *in* CA*, if there are* $t \in \pi$ *and* $t' \in \pi'$ *such that both* t *and* t' *fix* v *then* t *and* t' *assign the same sort to* v
2. CA *respects the (fixed) recursion context* ρ
3. *the underlying c-automaton obtained by removing the assertions from* CA *is deterministic*

is an asserted c-automaton *(ac-automaton for short).*

Intuitively, one can think of a variable v fixed at a transition t as "local" to the receiver of the interaction labelling t; also, the sender of the interaction is aware of the value to be assigned to v. Condition (1) in Def. 4.6 simply avoids confusion on the sort of a variable when it could be assigned along different paths.

Without loss of generality, we can assume that $\mathsf{var}(t) \cap \mathsf{bvar}(\mathsf{A}) = \emptyset$ for all transitions and predicates A of an ac-automaton; in fact, such condition can be enforced by simply renaming bound variables in predicates. Hereafter, we write $q \xrightarrow{\lambda} q'$ instead of $q \xrightarrow[\top]{\lambda} q'$.

4.2 Consistent choreography automata

Our interpretation of transitions as rely-guarantee relations requires some care. Indeed, for a transition t to be viable, participants involved in t must "know" the variables used in t. In particular, if t is an interaction variable then the sender and receiver in t must "know" the assertion in t and participants involved in an iteration should "know" the invariant of the loop. Before formalising this in the next definition, we introduce the auxiliary concept of *assertion of a path of an ac-automaton*, which yields the conjunction of all assertions in π while substituting recursive variables with actual values of recursive calls. Formally, if $t = q_1 \xrightarrow[\mathsf{A}]{\mathbf{r} \cdot \iota} q_2$ then $\nabla(t) = \iota$, otherwise $\nabla(t)$ is the empty substitution.

Then the *assertion of a path* π is defined as $\mathbb{A}(\pi) = \mathbb{A}_{\mathsf{id}}(\pi)$ where

$$\mathbb{A}_\iota(\varepsilon) \triangleq \top \qquad \text{and} \qquad \mathbb{A}_\iota(q \xrightarrow[\mathsf{A}]{\lambda} q' \ \pi) \triangleq \mathsf{A}\iota' \wedge \mathbb{A}_{\iota'}(\pi) \quad \text{with} \quad \iota' = \iota[\nabla(q \xrightarrow[\mathsf{A}]{\lambda} q')]$$

Namely, the assertion of a path is the conjunct of all the assertions of its transitions once the recursion parameters are updated with their actual values. We can now define the notion of *knowledge* of a variable.

[1] A path is *simple* if no state occurs twice on it.

▶ **Definition 4.7** (Knowledge). *Let* CA *be an ac-automaton. A participant* $\mathsf{p} \in \mathcal{P}$ knows v at a transition $t = q \xrightarrow[\mathsf{A}]{\lambda} q'$ in CA *if*

- *either t fixes v and*
  (a) *if $\lambda \in \mathcal{L}_{int}$ then $\mathsf{p} \in \mathit{ptp}(\lambda)$ and*
  (b) *if $\lambda = \mathbf{r} \cdot \iota$ with $\rho(\mathbf{r}) = (\mathtt{V}, \mathsf{A}, q')$ and p is on a cycle from q' to q' then $\mathsf{v} \in \mathtt{V}$*
- *or $\mathsf{v} \in \mathit{var}(\mathsf{A})$ and there are a variable u and a transition t' on each path $\pi \in \mathit{SPath}_{\mathsf{CA}}(q)$ such that p knows u at t' and $\mathbb{A}(\pi) \supset \mathsf{v} = \mathsf{u}$ holds.*

Let $\mathsf{knw}_{\mathsf{CA}}(\mathsf{p}, t)$ *be the set of variables that* p *knows at* t *in* CA*.*

▶ **Example 4.8** (OLW knowledge). In the FSA of Ex. 4.2 both vendor and customer know bill at the outgoing transition of state q_5. Also, customer and wallet know the recursion variable try of the ac-automaton in Ex. 4.3. ⌟

The notion of knowledge in Def. 4.7 is more complex than the one in [3]; this is an effect of the higher complexity in the notions of binding and scoping of variables. Def. 4.7 is instrumental to transfer the concept of *history-sensitivity* introduced in [3] to ac-automata.

▶ **Definition 4.9** (History sensitiveness). *An ac-automaton* CA *is* history-sensitive *if the following holds for each transition* $t = q \xrightarrow[\mathsf{A}]{\lambda} q'$ *in* CA

1. $\lambda = \mathsf{p} \to \mathsf{q} : \mathsf{m}$ *implies* $\mathit{var}(\mathsf{A}) \subseteq \mathsf{knw}_{\mathsf{CA}}(\mathsf{p}, t)$*, namely* p *knows each variable free in* A *at* t*.*
2. $\lambda = \mathbf{r} \cdot \iota$ *implies* $\mathit{var}(\mathbf{r}) \subseteq \mathsf{knw}_{\mathsf{CA}}(\mathsf{p}, t)$ *for each* $\mathsf{p} \in \mathcal{P}$ *occurring on a cycle from* q' *to* q'*.*

Condition (1) guarantees that the assertion of a transition cannot predicate on variables not “accessible” to the participants of the interaction. Condition (2) ensures that participants involved in a loop are aware of the loop invariant. The notion of history sensitivity in [3] relies on the fact that participant p knows a variable v on each interaction involving v. Here instead a weaker notion is adopted since, due to selective participation, the c-automaton may have a transition fixing v but not involving p.

▶ **Example 4.10** (OLW is history-sensitive). The ac-automaton in Ex. 4.3 is history-sensitive. In particular, note that the variable try in the assertion on the transition from q_2 to q_3 is known to customer and wallet since it is in the invariant of the authentication loop. ⌟

For a transition t of an ac-automaton CA to be enabled, it is not enough that the source state of t is reachable from the initial state of CA. In fact, the transition t can be fired if the information accumulated by the participants ensures the satisfiability of the assertion of t. To formalise this notion we introduce the following definitions. Given a state q of an ac-automaton CA, we let

$$\mathbb{P}_{\mathsf{CA}}(q) \triangleq \{\mathbb{A}(\pi) \mid \pi \text{ run to } q \text{ in } \mathsf{CA} \text{ and } \mathbb{A}(\pi) \text{ is satisfiable}\} \tag{2}$$

be the set of *preconditions* of q (in CA) and

$$\mathbb{E}_{\mathsf{CA}}(q) \triangleq \bigcup_{\mathsf{B} \in \mathbb{P}_{\mathsf{CA}}(q)} \left\{ \mathsf{B} \supset \bigvee_{q \xrightarrow[\mathsf{A}]{\lambda} q' \in \mathsf{CA}} \exists \mathsf{var}(\lambda) : \mathsf{A} \right\} \tag{3}$$

be the set of *enabling conditions* of q (in CA)

Similarly to [3] for global types, progress of ac-automata cannot be guaranteed if there is a possible computation leading to a state with no enabled transitions. Hence, we adapt from [3] the notion of *temporal satisfiability*.

▶ **Definition 4.11** (Temporal satisfiability). *An ac-automaton* CA *is* temporally satisfiable *if for each* $q \in \mathsf{CA}$ *reachable from the initial state of* CA *each formula in* $\mathbb{E}_{\mathsf{CA}}(q)$ *is satisfiable.*

▶ **Example 4.12** (OLW is temporally satisfiable). The ac-automaton in Ex. 4.3 is temporally satisfiable because the enabling conditions of all the nodes are satisfiable. However, if the assertion on the transition from q_2 to q_3 were replaced by e.g., $\mathsf{try} > 3 \wedge \mathsf{msg} = \text{"fail"}$ then temporal satisfiability would be violated because the precondition of the simple path from q_0 to q_2 would not entail $0 \leq \mathsf{try} < 3 \vee \mathsf{try} > 3$. ⌟

As c-automata, ac-automata are well-formed if they are well-sequenced and well-branched; these two notions are as for c-automata modulo the presence of assertions, which are disregarded; see [16] for the formal definitions. Finally, we can define *consistent* ac-automata.

▶ **Definition 4.13** (Consistency). *An ac-automaton is* consistent *if it is history-sensitive, temporally satisfiable, and well-formed.*

4.3 Asserted communicating systems

Projecting ac-automata requires to handle asserted transitions. We therefore extend communicating systems to *asserted communicating systems* (a-CSs for short), which basically are communicating systems where CFSMs are *asserted* (a-CFSMs for short), namely they have transitions decorated with formulae in $\mathcal{A}$. The synchronous semantics of a-CSs can be defined as an LTS similarly to the semantics of communicating systems. In fact, configurations can be defined as in Def. 2.9 taking into account assertions when synchronising transitions. This basically means that assertions are used to verify that a sent message guarantees the expectation of its receiver, that is the assertion the receiver relies upon.

Recall that a *prenex normal form* is a formula $\mathcal{Q}\mathsf{A}$ where $\mathcal{Q}$ is a sequence of quantifiers and variables (called *prefix*) and A is a quantifiers-free logical formula (called *matrix*) [34]. If $\mathsf{A}, \mathsf{B} \in \mathcal{A}$ then $\mathsf{A} \circ \mathsf{B}$ is a logical formula obtained by quantifying with the prefix of a prenex normal form A' logically equivalent to A the conjunction of B with the matrix of A'. Similarly to assertions for paths on ac-automata, we define assertions of a run of an a-CFSM

$$\mathbb{A}(\varepsilon) \triangleq \top \qquad \text{and} \qquad \mathbb{A}(q \xrightarrow[\mathsf{A}]{\ell} q' \ \pi) \triangleq \mathsf{A} \circ \mathbb{A}(\pi)$$

The preconditions of a state of an a-CFSM are defined as for ac-automata but for the use of the assertion function $\mathbb{A}$ for CFSMs instead of the corresponding one for ac-automata.

▶ **Definition 4.14** (Semantics of a-CS). *The* semantics *of an a-CS* $\mathsf{S} = (M_{\mathsf{p}})_{\mathsf{p} \in \mathcal{P}}$ *is the transition system* $[\![\mathsf{S}]\!]$ *defined by taking the set of configurations as in Def. 2.9 and as set of transitions the smallest set including*

- $s_1 \xrightarrow[\mathsf{A}]{\mathsf{p} \to \mathsf{q}:\mathsf{m}} s_2$ *if* $\mathsf{p}, \mathsf{q} \in \mathcal{P}$ *and*
  - $s_1(\mathsf{p}) \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} s_2(\mathsf{p})$ *in* M_{p}, $s_1(\mathsf{q}) \xrightarrow[\mathsf{B}]{\mathsf{pq?m}} s_2(\mathsf{q})$ *in* M_{q} *and, there are* $\mathsf{A}' \in \mathbb{P}_{M_{\mathsf{p}}}(s_1(\mathsf{p}))$ *and* $\mathsf{B}' \in \mathbb{P}_{M_{\mathsf{q}}}(s_1(\mathsf{q}))$ *such that it holds* $(\mathsf{A}' \supset \mathsf{A}) \wedge (\mathsf{B}' \supset \mathsf{B}) \wedge (\mathsf{A}' \circ \mathsf{B}' \circ \mathsf{A}) \supset \exists \mathit{var}(\mathsf{m}) : \mathsf{B}$
  - *and* $s_1(\mathsf{x}) = s_2(\mathsf{x})$ *for all* $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p}, \mathsf{q}\}$
- $s_1 \xrightarrow[\mathsf{A}]{\varepsilon} s_2$ *if* $\mathsf{p} \in \mathcal{P}$ *and*
  - $s_1(\mathsf{p}) \xrightarrow[\mathsf{A}]{\varepsilon} s_2(\mathsf{p})$ *in* M_{p} *and there is* $\mathsf{A}' \in \mathbb{P}_{M_{\mathsf{p}}}(s_1(\mathsf{p}))$ *such that* $\mathsf{A}' \supset \mathsf{A}$
  - *and* $s_1(\mathsf{x}) = s_2(\mathsf{x})$ *for all* $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p}\}$.

Like the projection of communicating systems (cf. Def. 2.7), the projection of a-CSs relies on the determinisation and minimisation of a-CFSMs. The presence of assertions imposes to adapt the classical constructions on FSA to a-CFSMs. More precisely, we have to generalise equality on labels of the form (λ, A). Essentially, this is done by (injectively) renaming the variables occurring in actions and assertions decorating transitions. For σ an endofunction on variables and $\mathsf{m} = \tau \langle \mathsf{v}_1 \ \mathsf{s}_1, \dots, \mathsf{v}_h \ \mathsf{s}_h \rangle$ let $\mathsf{m}\sigma \triangleq \tau \langle \sigma(\mathsf{v}_1) \ \mathsf{s}_1, \dots, \sigma(\mathsf{v}_h) \ \mathsf{s}_h \rangle$; we define

$$\varepsilon\sigma \triangleq \varepsilon \qquad \text{and} \qquad (\mathsf{p} \to \mathsf{q} : \mathsf{m})\sigma \triangleq \mathsf{p} \to \mathsf{q} : \mathsf{m}' \qquad \text{where} \qquad \mathsf{m}' = \mathsf{m}\sigma$$

Two labels (λ, A) and (λ', A') are *equivalent*, in symbols $(\lambda, \mathsf{A}) \sim (\lambda', \mathsf{A}')$, if there is an injective substitution of variables such that $\lambda = \lambda'\sigma$ and A is logically equivalent to $\mathsf{A}'\sigma$. We will similarly consider equivalence on $\mathcal{L}_{\text{act}} \times \mathcal{A}$.

The ε-closure of an a-CFSM $M = (Q, q_0, \mathcal{L}_{\text{act}}, \mathcal{T})$ is the map $\varepsilon\text{-clos}_M : Q \to 2^{Q \times \mathcal{A}}$ defined assigning to each state q of M the set of states reachable with ε-transitions together with their assertions; more precisely, for each $q \in Q$, $\varepsilon\text{-clos}_M(q)$ is the smallest set satisfying

$$\varepsilon\text{-clos}_M(q) \triangleq \{(q, \top)\} \cup \bigcup_{(q', \mathsf{A}) \in \varepsilon\text{-clos}_M(q)} \left\{ (q'', \mathsf{A} \circ \mathsf{A}') \mid q' \xrightarrow[\mathsf{A}']{\varepsilon} q'' \in \mathcal{T} \right\} \tag{4}$$

Removal of ε-transitions from an a-CFSM M is computed, using (4), similarly to the classical algorithm on FSAs $\langle \mathbb{Q}, \varepsilon\text{-clos}(q_0), \mathcal{L}_{\text{act}}, \mathbb{T} \rangle$ where

$$\mathbb{Q} = \{\varepsilon\text{-clos}_M(q) \mid q \in Q\} \qquad \text{and}$$
$$\mathbb{T} = \{Q \xrightarrow[\mathsf{A}_1 \circ \mathsf{A} \circ \mathsf{A}_2]{\ell} Q' \mid q_1 \xrightarrow[\mathsf{A}]{\ell} q_2 \in \mathcal{T} \text{ for some } (q_1, \mathsf{A}_1) \in Q \text{ and } (q_2, \mathsf{A}_2) \in Q'\}$$

Handling assertions in the determinisation algorithm requires some care. We illustrate the problem in the following example.

▶ **Example 4.15** (Non-determinism & assertions). Consider the two a-CFSMs below

If both A and B are satisfiable then M has a non-deterministic behaviour. We therefore aim to define a determinisation algorithm which on M yields something like M'. Also, the new state q' should provide transitions corresponding to both transitions from q_1 and q_2. ⌟

Let $M = (Q, q_0, \mathcal{L}_{\text{act}}, \mathcal{T})$ be a CFSM. A state $q \in Q$ *is non-deterministic on* $\ell \in \mathcal{L}_{act}$ if its *derivative in* M *with respect to* ℓ, defined as $\partial_M(q, \ell) \triangleq \{(\mathsf{A}, q') \mid q \xrightarrow[\mathsf{A}]{\ell} q' \text{ in } M\}$, has more than one element. Also, if $X, Y \subseteq \partial_M(q, \ell)$ then $\Delta(X, Y) \triangleq \bigwedge_{(\mathsf{A}, q) \in X} \mathsf{A} \wedge \bigwedge_{(\mathsf{B}, q) \in Y} \neg \mathsf{B}$. The determinisation of M is obtained by applying the classical FSA determinisation algorithm to the ε-closure of the a-CFSM $M' = (Q', q_0, \mathcal{L}_{\text{act}}, \mathcal{T}' \cup \mathcal{T}'' \cup \mathcal{T}''')$ where

$$Q' \triangleq Q \cup \bigcup_{q \in Q, \ell \in \mathcal{L}_{\mathrm{act}}} \{ \langle X \rangle \mid q \text{ is non-deterministic on } \ell \text{ and } \emptyset \neq X \subseteq \partial_M(q,\ell) \}$$

$$\mathcal{T}' \triangleq \left\{ q \xrightarrow[\mathsf{A}]{\ell} q' \in \mathcal{T} \mid \partial_M(q,\ell) \text{ is a singleton} \right\}$$

$$\mathcal{T}'' \triangleq \bigcup_{\emptyset \neq X \subseteq \partial_M(q,\ell)} \left\{ q \xrightarrow[\Delta(X,Y)]{\ell} \langle X \rangle \mid \partial_M(q,\ell) \text{ not a singleton} \quad \text{and} \quad Y = \partial_M(q,\ell) \setminus X \right\}$$

$$\mathcal{T}''' \triangleq \bigcup_{\emptyset \neq X \subseteq \partial_M(q,\ell)} \left\{ \langle X \rangle \xrightarrow[\mathsf{A}]{\ell} q' \mid \text{there is } q \xrightarrow[\mathsf{A}]{\ell} q' \in \mathcal{T} \quad \text{with} \quad \{q\} \times \mathcal{A} \cap X \neq \emptyset \right\}$$

Basically, we (i) introduce a new state $\langle X \rangle$ for any combination of assertions of ℓ-transitions, (ii) replace non-deterministic behaviours on ℓ with a set of ℓ-transitions with "disjoint" assertions, and (iii) let state $\langle X \rangle$ have the transitions that any of the states $q \in X$ has in M.

We remark that the adaptation of the determinisation algorithm is imposed by the use of a-CFSMs to model local behaviour. This is a main technical difference with respect to [3] where local types with assertions, which need no determinisation, play the role of a-CFSMs.

The projection of an ac-automaton acts as the projection of c-automata on interactions and accommodates the variables not known to the participant by existentially quantifying them. This requires to consider the points in the ac-automaton where variables are fixed.

▶ **Definition 4.16** (Projection of ac-automata). *The* projection on $\mathsf{p} \in \mathcal{P}$ *of an asserted transition t in an ac-automaton* CA *on* $\mathcal{P}$, *written* $t{\downarrow_{\mathsf{CA},\mathsf{p}}}$, *is defined by:*

$$t{\downarrow_{\mathsf{CA},\mathsf{p}}} = \begin{cases} q \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} q' & \text{if } t = q \xrightarrow[\mathsf{A}]{\mathsf{p \to q{:}m}} q' \\ q \xrightarrow[\mathsf{A}]{\mathsf{qp?m}} q' & \text{if } t = q \xrightarrow[\mathsf{A}]{\mathsf{q \to p{:}m}} q' \\ q \xrightarrow[\exists X :\ \mathsf{A}(\nabla(t))]{\varepsilon} q' & \text{if } t = q \xrightarrow[\mathsf{A}]{\lambda} q',\ \mathsf{p} \notin ptp(\lambda),\ \text{and } X = \{ v \in var(\mathsf{A}) \mid t \text{ fixes } v \text{ in } \mathsf{CA}\} \end{cases}$$

The projection *of* CA *on* $\mathsf{p} \in \mathcal{P}$, *denoted* $\mathsf{CA}{\downarrow_{\mathsf{p}}}$, *is obtained by determinising and minimising up-to-language equivalence the* intermediate *a-CFSM*

$$A_{\mathsf{p}} = \left\langle \mathbb{S}, q_0, \mathcal{L}_{act}, \left\{ (q \xrightarrow[\mathsf{A}]{\lambda} q'){\downarrow_{\mathsf{CA},\mathsf{p}}} \mid q \xrightarrow[\mathsf{A}]{\lambda} q' \text{ in } \mathsf{CA} \right\} \right\rangle$$

where (i) syntactic equality of labels is replaced by $\sim$ and (ii) ε-transitions are those with label of the form $(\varepsilon, \mathsf{A})$. The projection of CA, *written* $\mathsf{CA}{\downarrow}$, *is the a-CS* $(\mathsf{CA}{\downarrow_{\mathsf{p}}})_{\mathsf{p} \in \mathcal{P}}$.

We show that projections of consistent ac-automata yield deadlock-free asserted communicating systems. The next result corresponds to Thm. 3.12 for ac-automata. The main differences are (i) that consistency of ac-automata is required (as opposed to well-formedness for c-automata) and (ii) that an ac-automaton is weakly bisimilar to the corresponding projected system due to the fact that iterative transitions of the ac-automaton are projected on ε-transitions.

▶ **Proposition 4.17.** *Any consistent ac-automaton* CA *is weakly bisimilar to* $[\![\mathsf{CA}{\downarrow}]\!]$.

As for c-automata, Prop. 4.17 ensures that the language of a consistent ac-automaton coincides with the language of its projection.

▶ **Corollary 4.18.** $L(\mathsf{CA}) = L([\![\mathsf{CA}{\downarrow}]\!])$ *for any consistent ac-automaton* CA.

Final states and deadlock freedom of an ac-automaton are defined as for c-automata (cf. Def. 3.14 and Def. 3.15 respectively) modulo the different labels of transitions.

▶ **Theorem 4.19** (Projections of consistent ac-automata are deadlock-free)**.** *If* CA *is a consistent ac-automaton then* CA↓ *is deadlock-free.*

Observe that Thm. 4.19 requires ac-automata to be consistent; in particular, it requires history sensitiveness (cf. Def. 4.9) and temporal satisfiability (cf. Def. 4.11). The two requirements ensure that assertions on the transitions do not spoil deadlock freedom.

5 TypeScript Programming via Flexible C-Automata

We showcase the main theoretical results and constructions in this paper with a tool, CAScr, the first implementation of Scribble [21, 38, 48] that relies on c-automata, for deadlock-free distributed programming. CAScr takes the popular *top-down approach* to system development based on choreographic models, following the original methodology of Scribble and multiparty session types [22]. The top-down approach enables *correctness-by-construction*: a developer provides a global description for the whole communication protocol; by projecting the global protocol, APIs are generated from local CFSMs, which ensure the safe implementation of each participant. The core theory of c-automata from § 3 guarantees deadlock freedom for the distributed implementation of flexible global protocols. As a first application we target web development, supporting in particular the TypeScript programming language.

In this section we present our development in three steps:

1. *translation of global protocols into choreography automata*: for the specification of global protocols, CAScr relies on the Scribble language, and global Scribble protocols are formally global multiparty session types protocols [38]; we define a function that maps these into choreography automata, and discuss the relation between the two formalisms;
2. *protocol specification and projections*: from the specification of the global protocol, CAScr generates, through its translation into c-automata and the subsequent projection, a collection of CFSMs, which are the abstract representation of the communication behaviour of each participant (cf. part **(a)**, Fig. 3a on page 21);
3. *API generation for deadlock-free distributed web development*: we discuss our choice of targeting TypeScript and web development, and illustrate how CAScr provides support for this (cf. part **(b)**, Fig. 3a on page 21); finally we comment on possible extensions.

5.1 From Multyparty Session Protocols to C-Automata

C-automata and asserted c-automata can be directly produced by the system designer and fed to our approach to ensure their correct behaviour. However, to improve the usability of the approach, our implementation, detailed in the next section, integrates c-automata with the Scribble framework. This framework is based on the theory of global types, hence we study below the relations between global types and c-automata. The syntax of global types is given by the following grammar:

$$G \quad ::= \quad \mathsf{end} \quad | \quad \mu \mathbf{r}.G \quad | \quad \mathbf{r} \quad | \quad \mathbf{\Sigma}_{i\in I}\,\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i$$

We simply write $\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i$ instead of $\mathbf{\Sigma}_{i\in I}\,\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i$ when $I = \{i\}$. In a recursive type $\mu\mathbf{r}.G$ all occurrences of the recursion variable $\mathbf{r}$ in G are bound (this is the only binder for global types); we moreover assume that the occurrences of $\mathbf{r}$ in G are guarded. Hereafter we assume the so-called Barendregt convention, that is names of bound variables are all distinct and different from names of free variables.

$$[\textsc{Choice}]\ \mathbf{\Sigma}_{i\in I}\mathsf{p}\to\mathsf{q_i}:\mathsf{m_i};G_i \xrightarrow{\mathsf{p}\to\mathsf{q_j}:\mathsf{m_j}} G_j \quad (j\in I)$$

$$[\textsc{Rec}]\ \frac{G[\mu\mathbf{r}.G/\mathbf{r}] \xrightarrow{\alpha} G'}{\mu\mathbf{r}.G \xrightarrow{\alpha} G'} \qquad [\textsc{Pass}]\ \frac{G_j \xrightarrow{\alpha} G'_j \quad \mathsf{p},\mathsf{q}_j\notin\mathsf{ptp}(\alpha) \quad \forall j\in I}{\mathbf{\Sigma}_{i\in I}\mathsf{p}\to\mathsf{q_i}:\mathsf{m_i};G_i \xrightarrow{\alpha} \mathbf{\Sigma}_{i\in I}\mathsf{p}\to\mathsf{q_i}:\mathsf{m_i};G'_i}$$

Figure 2 LTS semantics over global types.

The operational semantics of global types is the LTS induced by the rules in Fig. 2 where labels are drawn from the alphabet $\mathcal{L}_{\text{int}}$. Since the semantics of global types is an LTS, it can be represented as a c-automaton only if it is finite state. Unfortunately, the interplay between rule [Pass] and recursion allows one to generate infinite state LTSs, as shown below.

▶ **Example 5.1** (Infinite-state LTS). Let $G_{\text{inf}} = \mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r}+\beta.\mathsf{end})$ where $\delta \parallel \alpha$, $\delta\parallel\gamma$, $\alpha\not\parallel\gamma$, $\beta\not\parallel\delta$, and $\beta\not\parallel\alpha$. Note that the traces $(\alpha\ \gamma)^n$ are included in the semantics for all $n > 1$. Executing $(\alpha\ \gamma)^n$ results in the following computation:

$$\mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r}+\beta.\mathsf{end}) \xrightarrow{\alpha} \alpha;\mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r})+\gamma;\delta;\mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r})+\beta.\mathsf{end}$$
$$\xrightarrow{\gamma} \delta;\mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r}+\beta.\mathsf{end}) \quad \ldots \quad \xrightarrow{\gamma} \delta^n;\mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r}+\beta.\mathsf{end})$$

States $\delta^n;\mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r}+\beta.\mathsf{end})$ and $\delta^m;\mu\mathbf{r}.\alpha;(\alpha;\mathbf{r}+\gamma;\delta;\mathbf{r}+\beta.\mathsf{end})$ are bisimilar only if $n = m$. Indeed, one needs to execute n times δ (and an α) before being able to execute β. ⌟

It is worth remarking that the semantics in Fig. 2 yields finite-state LTSs on global types without consecutive independent transitions, a restriction actually considered in many global type formalisms, since rule [Pass] never applies. Likewise, the semantics consisting of rules [Choice] and [Rec] only generates finite-state LTSs.

Function $\mathsf{ca}(G)$ below defines a c-automaton with subterms of G as states, G as initial state, labels in $\mathcal{L}_{\text{int}}$, and transitions inductively defined by the function $\mathsf{catr}(G)$ below:

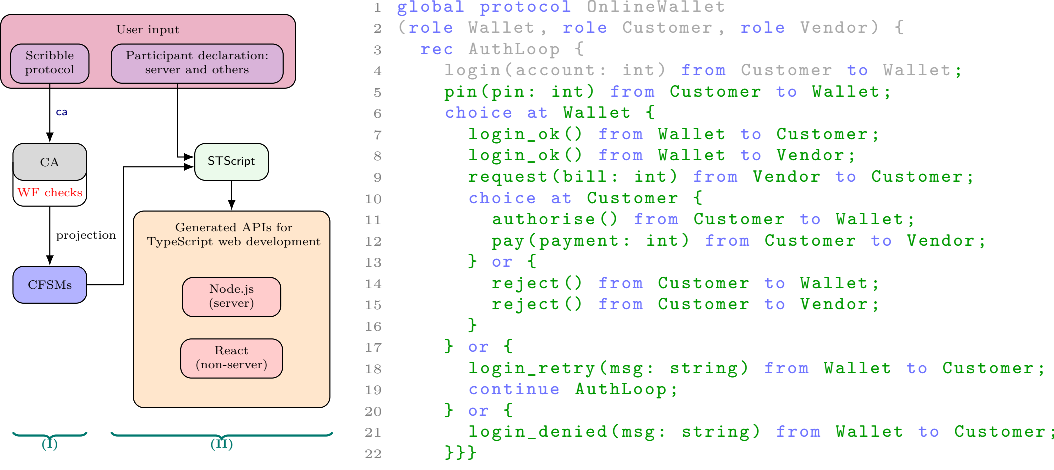
$$\mathsf{catr}(\mathsf{end}) = \mathsf{catr}(\mathbf{r}) = \emptyset \qquad \mathsf{catr}(\mu\mathbf{r}.G) = \mathsf{catr}(G)\cup\{(\mathbf{r},\epsilon,\mu\mathbf{r}.G),(\mu\mathbf{r}.G,\epsilon,G)\}$$
$$\mathsf{catr}(\mathbf{\Sigma}_{i\in I}\mathsf{p}\to\mathsf{q_i}:\mathsf{m_i};G_i) = \bigcup_{j\in I}(\{(\mathbf{\Sigma}_{i\in I}\mathsf{p}\to\mathsf{q_i}:\mathsf{m_i};G_i,\mathsf{p}\to\mathsf{q_j}:\mathsf{m_j},G_j)\}\cup\mathsf{catr}(G_j))$$

▶ **Proposition 5.2.** *Let G a global type. The language of $\mathsf{ca}(G)$ coincides with the language generated by rules [Choice] and [Rec] of the semantics of G.*

Function $\mathsf{caPass}(G)$ below extends $\mathsf{ca}(G)$ to deal with the semantics of global types with rule [Pass]. However, the computed LTS may be infinite state, hence not a c-automaton, and in this case the function cannot be used in practice. This is, e.g., the case with the global type in Ex. 5.1. The LTS has G as initial state, labels in $\mathcal{L}_{\text{int}}$, transitions inductively defined by the function $\mathsf{catrPass}(G)$ below, and as states the ones occurring in the transitions:

$$\begin{aligned}
\mathsf{catrPass}(\mathsf{end}) = \mathsf{catrPass}(\mathbf{r}) &= \emptyset\\
\mathsf{catrPass}(\mu\mathbf{r}.G) &= \mathsf{catrPass}(G)\cup\{(\mathbf{r},\epsilon,\mu\mathbf{r}.G),(\mu\mathbf{r}.G,\epsilon,G)\}\\
\mathsf{catrPass}(\mathbf{\Sigma}_{i\in I}\alpha_i;G_i) &= \bigcup_{j\in I}(\{(\mathbf{\Sigma}_{i\in I}\alpha_i;G_i,\alpha_j,G_j)\}\cup\mathsf{catrPass}(G_j))\cup\\
&\qquad \bigcup_{\alpha\text{ s.t. } G_i\xrightarrow{\alpha}G'_i\wedge\alpha_i\parallel\alpha\forall i\in I}\{(\mathbf{\Sigma}_{i\in I}\alpha_i;G_i,\alpha,\mathbf{\Sigma}_{i\in I}\alpha_i;G'_i)\}\cup\mathsf{catrPass}(\mathbf{\Sigma}_{i\in I}\alpha_i;G'_i)
\end{aligned}$$

▶ **Proposition 5.3.** *Let G a global type. The language of $\mathsf{caPass}(G)$ coincides with the language generated by the semantics of G.*




```
global protocol OnlineWallet
(role Wallet, role Customer, role Vendor) {
  rec AuthLoop {
    login(account: int) from Customer to Wallet;
    pin(pin: int) from Customer to Wallet;
    choice at Wallet {
      login_ok() from Wallet to Customer;
      login_ok() from Wallet to Vendor;
      request(bill: int) from Vendor to Customer;
      choice at Customer {
        authorise() from Customer to Wallet;
        pay(payment: int) from Customer to Vendor;
      } or {
        reject() from Customer to Wallet;
        reject() from Customer to Vendor;
      }
    } or {
      login_retry(msg: string) from Wallet to Customer;
      continue AuthLoop;
    } or {
      login_denied(msg: string) from Wallet to Customer;
    }}}
```

(a) Toolchain of CAScr. **(b)** Scribble Protocol for the OLW.

Figure 3 CAScr: Toolchain and OLW Protocol.

We remark that global types with infinite semantics cannot be implemented faithfully using communicating systems with the semantics in Def. 2.9. Indeed, a communicating system has a finite number of configurations, which is $O(S^n)$ where S is the size of the largest CFSM and n the number of participants.

5.2 Validating Global Protocols with Choreography Automata

The first component of our toolchain is part **(I)** in Fig. 3a; it allows the user to perform protocol specification, well-formedness checks, and the generation of CFSMs for each participant.

Let us consider the OLW example: the first step for the user is to specify the global protocol, `OnlineWallet.scr` (Fig. 3b), in the *Scribble protocol description language*, often referred to as "the practical incarnation of multiparty session types" [21, 38]. The syntax of Scribble (`http://www.scribble.org`, `https://nuscr.dev/`) has a straightforward correspondance to the syntax of global types, so Scribble implementations of communicating processes will be supported by multiparty session type theory, and inherit its semantic guarantees. Our development for part **(I)** of the toolchain is based on the νScr implementation [48], but fundamentally differs from this (and other Scribble versions) in two aspects:

- the underlying choreographic objects – normating the communication among multiple participants – are not global types, but c-automata, and
- we allow for participants to join the communication at later stage, only in branches where they are needed (selective participation).

Fig. 3b shows the protocol `OnlineWallet.scr` for the OLW. Noteworthy, unlike νScr, we can specify the selective participation of the vendor. In particular, the vendor participant is involved only in the first branch of the choice (lines 7-12), namely on successful login.

After its specification, the Scribble protocol is translated into a c-automaton, with the implementation of the function `ca` from § 5.1 (this is exactly the c-automaton from Ex. 2.5, § 2). On this automaton, well-sequencedness and well-branchedness checks are performed. If the c-automaton passes the above well-formedness checks, it is then projected onto each participant (Def. 2.7), thus obtaining a collection of CFSMs, whose semantics is equivalent to the one of the original c-automaton. Both global c-automata and local CFSMs are represented

```
17  const onlineWalletServerLogic = (sessionID: string) => {
18      DB.initSession(sessionID);
19      const handleRequest = Session.Initial({
20          login: (Next, accountPayload) =>
21              Next({
22                  pin: (Next, pinPayload) => {
23                      if (accountPayload.account === 100000 && pinPayload.pin === 1000) {
24                          console.log('Login details correct!');
25                          return Next.
26  (property) login_denied: {        ×    login_denied
27      (payload: Message.S2_login_den…     login_ok
28      (payload: Message.S2_login_den…     login_retry
29  }
30
```

Figure 4 Implementation with the API for wallet in Visual Studio Code.

using the DOT graph description language. Ex. 2.8 from § 2 shows the CFSM obtained by projection on vendor of the c-automaton for the OLW; for the other participants analogous CFSMs are obtained. The local CFSM representations provide the communication behaviour of each participant and, as such, they retain all the information for obtaining deadlock-free endpoint implementations. Each CFSM is the projection of the global c-automaton onto one of the communicating participants; from this local automaton, the API for the implementation of the participant is generated.

5.3 API Generation for Distributed Web Development

Our chosen domain of application is *distributed web development*. By nature, web services are distributedly developed and feature communication among multiple participants. In services where some courses of actions are optional, it is likely that the participation of some role is also optional (selective participation). Our OLW example is a minimal, yet representative example that selective participation is commonplace in transactions, auctions, or contracts. For instance, Kickstarter [30] is a worldwide popular crowdfunding platform where the money of *supporters* is given to a project *initiator* only if the initially set goal is met; otherwise the money is returned to supporters. In other words, when the deadline is passed, if the goal is met, only the initiator is involved in the communication, if not, only the supporters are.

More technically, our development builds on and extends STScript [36]. We target server-centric protocols (based on the WebSocket standard [13]), where one role is chosen as privileged, the *server*. The generated APIs are compatible with the Node.js runtime for server-side endpoints and the React.js framework for browser-side endpoints. The STScript toolchain in [36] is based on the multiparty session type theory, where there is no privileged role; hence which role is the server has to be declared by the user. The same holds for our development based on c-automata. We have discussed in the previous section how the Scribble protocol in input is translated into a c-automaton and, once well-formedness checks are performed, projected onto a CFSM for each participant. This CFSM is passed to the code-generation component of our toolchain (part **(II)** of Fig. 3a), together with the role in input and the information about whether it is the server role or not.

Fig. 4 shows an example of the usage of the generated API, when implementing the participant wallet in Visual Studio Code (`https://code.visualstudio.com/`). The autocomplete function of the editor offers the developer appropriate options, so that the implementation of the login choice abides by the global discipline of the OnlineWallet protocol.

From an engineering point of view, for developing the part **(I)** of the toolchain (Fig. 3a) we have adapted to our theory of c-automata, the codebase of νScr: a recent implementation of Scribble that offers a toolchain for "language-independent code generation" [48]. However,

νScr itself does not provide direct support for TypeScript. Hence, the development of part **(II)** in Fig. 3a integrates the νScr codebase with STScript. This is is a Scribble extension – also based on multiparty session types, but relying on the ScribbleJava implementation `http://www.scribble.org`. Building the API-generation of CAScr on top of the one of STScript has been a conventient choice: STScript targets distributed web development directly and offers a full implementation for generating TypeScript APIs from νScr-projected CFSMs.

The result of our development is CAScr, of which we list the distinctive features.

- *Scope.* CAScr specifically targets TypeScript and enables safe distributed web development.
- *Input.* The user specifies the global protocol in the Scribble language and picks one of the communicating participants as the server.
- *Correctness.* CAScr relies on the flexible theory of c-automata: the protocol in input is translated into a c-automaton, which, if well-formed, is then projected onto CFSMs.
- *APIs Generation.* From each CFSM, CAScr generates the TypeScript API for the respective role.
- *Safe Endpoint Implementation.* The distributed implementation of the participants, using the generated APIs, is guaranteed to be deadlock and lock free by the underlying theory.

In our first implementation of CAScr (`https://github.com/Tooni/CAScript-Artifact`), we provide three simple examples: an "adder" (the client sends to the server, in a loop, two numbers to be added), a simple contract protocol, and the OLW, which we have used as a running example, since it carries and shows all the core features of our novel theory, and, in particular, selective participation (see also the discussion at the beginning of this section). Furthermore, we have provided a small tutorial in the README file of CAScr, to guide the user through the implementation of their own protocols.

It is worth mentioning that a first extension of CAScr is under development (see also [16]): current implementation, based on previous work [49], allows the generation of APIs for Scribble protocols with assertions. However, the necessary extension of the function ca in § 5.1 to assertions, as well as subsequent consistency checks, have not been implemented yet. While conceptually straightforward, in practice one needs to integrate the CAScr toolchain with tools manipulating logical formulae such as SAT solvers in order to implement the check for the consistency property (cf. Def. 4.13).

To conclude, we have developed the first version of Scribble based on *choreography automata*. It improves on the flexibility of traditional implementations of multiparty session types, by accomodating for *selective participation*, and it integrates previous developments with our new theory: the νScr toolchain with the TypeScript support provided by STScript. On the one hand, our toolchain enables verified communication for web development with selective participation, on the other hand it paves the road to interesting extensions, e.g., fully capturing the asynchronous semantics of websockets (see § 7), or supporting assertions and the design-by-contract approach, as discussed above.

6 Related Work

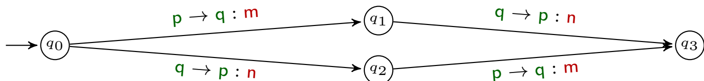
Conditions similar to our well-branchedness and well-sequencedness arise naturally in investigations on choreographies and their realisability. Uniqueness of choice selector is commonly imposed syntactically (as in § 5.1) in several multiparty session types (MPSTs) formalisms (e.g., [22, 3, 9, 45, 49]) and also adopted in global graphs [11, 46], and in choreography languages in general (cf. the notion of *dominant role* in [41]). Also, notions close to well-sequencedness occur quite naturally in "well-behaved" choreographies (e.g., the notion of *well-informedness* of [6] in collaboration diagrams). A distinguishing element of our notion of

well-branchedness is that we admit protocols where disjoint groups of participants may concurrently engage in a choice. This generalises (and corrects) the notion of well-branchedness in [2] and, to the best of our knowledge, is not supported in any other choreographic framework.

Global graphs [11, 17, 46, 33] are another model of global specifications. We refer the reader to [2] for a comparison between c-automata and global graphs.

The first work advocating a design-by-contract framework for MPSTs is [3]. Asserted c-automata have been strongly inspired by it. In particular, our notion of consistency (cf. Def. 4.13) can be seen as a generalisation of *well-assertedness* in [3]. More recently, ideas similar to the one in [3] have been developed in [49], where refined MPSTs have been proposed. The results of these papers are in the vein of guaranteeing properties of programs by a behavioural type system ensuring communication soundness in presence of data dependencies.

Besides the added flexibility of c-automata with respect to structured formalisms discussed in the Introduction, ac-automata do not suffer from the constraints imposed on global types in [3, 49]. More precisely, interactions guarding choices in [3, 49] syntactically restrict to a unique partner of the *selector* (i.e., the participant choosing the branch to follow). On the contrary, (asserted) c-automata do not have such restriction. For instance,



is a well-branched c-automaton which would be ruled out by all the choreography models based on global types we are aware of. Both [3, 49] rely on a merge operator to guarantee well-formedness (and projectability) of global types. This is an obstacle for selective participation which our notion of well-branchedness (cf. Def. 3.7) overcomes. We also note that our notion of knowledge is more general than the one in [3]. In fact, as observed in [49], the notion of history sensitivity in [3] does not allow a participant to know variables fixed in interactions it is not involved in. Like for refined MPSTs, asserted c-automata do not have this limitation and can in fact deal with protocols like the one in Example 4.1 in [49].

Our theoretical work sees its first application in the development of CAScr, a toolchain for communication-safe web development. CAScr takes the popular *top-down approach*, following the original methodology of MPSTs [22]. The top-down approach enables *correctness-by-construction*: a developer provides a global description for the whole protocol; by projecting the global protocol, APIs are generated from local CFSMs, which ensure the safe implementation of each participant. MPSTs toolchains that take the top-down approach have seen multiple implementations and targeted a variety of mainstream programming languages, such as (in no particular order) Java [24, 25, 31], OCaml [27], Go [8], Scala [43, 47], F# [37], F* [49] and Rust [10, 32]. Like CAScr, most of the above implementations rely on the Scribble protocol description language [21, 38, 48] (`http://www.scribble.org`, `https://nuscr.dev/`). More relevant to this work is [36], in which the authors develop STScript, a full toolchain that applies such top-down methodology and targets TypeScript for web development.

All the implementations above are based on MPSTs; they exploit the equivalence between local types and CFSMs [11, 12] to generate APIs for all the participants. In [25], *explicit connections*, similar to our selective participation, have been introduced in Scribble, and more recently [19] uses an analogous approach to implement adaptations for an actor domain-specific language. Both [25] and [19] need to add explicit disconnections and connections to the syntax of Scribble. In CAScr (§ 5), we have integrated the theory of c-automata into the νScr toolchain [48], to allow for more flexible protocols, where participants may appear only in selected branches after a choice, with no need to change the Scribble syntax.

7 Conclusion and Future Work

We have presented a flexible framework to describe protocols in a setting of c-automata combining selective participation to branches of choices and assertions supporting design-by-contract. This allows us to model non trivial examples such as the OnLineWallet, and ensures faithful realisability. In fact, we exploited the flexibility of c-automata to generalise well-branchedness (so to account for selective participation) and to transfer the DbC approach [3] (so to account for *data-aware* protocols). Remarkably, the fact that c-automata are finite-state models does not allow us to fully capture Scribble. Nonetheless, a semi-decidable approach has been considered (cf. § 5.1) which becomes effective when restricting to protocols without interplay between consecutive independent interactions and recursion. More precisely, it should not be possible to split a recursive protocol into groups of interactions with disjoint participants. This restriction mildly affects applicability: indeed, to faithfully implement such specifications one would need infinite-state systems of CFSMs, while ours are finite-state. Also, a clear advantage of our approach is that we can verify more general conditions for Scribble specifications that can be faithfully mapped on c-automata.

We implemented our theory by allowing Scribble protocols to be translated into c-automata, checked for well-formedness, and finally used to derive APIs for TypeScript programming. The flexibility of c-automata has been instrumental to capture Scribble [21, 38, 48] specifications. Scribble notation (and semantics) may be not easy to grasp for practitioners as it involves a non-trivial amount of technicalities. Hence, defining and understanding well-formedness conditions on Scribble could not be straightforward.

Our framework can be immediately used in practice in interesting examples: the design of a variety of existing web services (e.g., for authentication or transactions) include selective participation; with the OLW implementation, we witness how protocols carrying this feature can be specified in CAScr (which from these generates APIs for implementations). Nonetheless, we envisage some extensions (see § 5.3 and [16] for details).

Our focus is on selective participation and design-by-contract. Hence, for simplicity, we consider synchronous semantics. CAScr builds instead on an asynchronous implementation of Scribble [36], which makes our results applicable only to protocols in which asynchronous executions do not break the causal relations imposed by the synchronous semantics so that choices are affected. This is the case for the case studies in the artifact, including our running example OLW. The discrepancy disappears if a synchronous transport layer (e.g., http) replaces web sockets. To increase the applicability of CAScr – and also because of its theoretical interest, we plan to extend the results to cover an asynchronous communication model based on queues. While the general structure of the theory remains the same, well-branchedness needs to be updated since send and receive actions would not be symmetric anymore. E.g., a participant that only occurs in one branch of a choice, thanks to selective participation, needs to interact with a fully-aware participant by performing a receive, while right now it can also interact through a send action. We conjecture that the extension to asynchronous semantics does not affect the treatment of DbC in ac-automata. In fact, assertions are guaranteed by the sender and relied upon by the receiver (hence, the nature of communication is orthogonal to the flow of data).

Our methodology follows the top-down software development approach of choreographies (cf. § 1 and § 6). An interesting direction for future work is to develop an analysis of existing APIs; for instance, by extracting an abstract representation of the API, its conformance could be checked against a projection of the global specification. Such design would improve on the applicability of our theory, for analysing and reusing existing developments.

References

1. Marco Autili, Paola Inverardi, and Massimo Tivoli. Automated synthesis of service choreographies. *IEEE Softw.*, 32(1):50–57, 2015. doi:10.1109/MS.2014.131.
2. Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020.
3. Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In Paul Gastin and François Laroussinie, editors, *Concur 2010*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
4. Jonas Bonér. *Reactive Microsystems - The Evolution Of Microservices At Scale*. O'Reilly, 2018.
5. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
6. Tevfik Bultan and Xiang Fu. Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008. doi:10.1007/s11761-008-0022-7.
7. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.
8. David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in Go: Statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290342.
9. Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.
10. Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693 of *PPoPP '22*, pages 261–246. ACM, 2022. doi:10.1145/3503221.3508404.
11. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. doi:10.1007/978-3-642-28869-2_10.
12. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming*, pages 174–186, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
13. Ian Fette and Alexey Melnikov. The websocket protocol, 2011. URL: https://www.rfc-editor.org/info/rfc6455.
14. Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
15. Leonardo Frittelli, Facundo Maldonado, Hernán C. Melgratti, and Emilio Tuosto. A choreography-driven approach to APIs: The OpenDXL case study. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2020. doi:10.1007/978-3-030-50029-0_7.
16. Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-by-contract for *Flexible* multiparty session protocols – extended version. Technical report, Focus Team, University of Bologna/INRIA (Italy) and Gran Sasso Science Institute (Italy) and Imperial College (UK), May 2022. Full version of the ECOOP 2022 paper. URL: http://mrg.doc.ic.ac.uk/publications/design-by-contract-for-flexible-multiparty-session-protocols/.

17 Roberto Guanciale and Emilio Tuosto. Realisability of pomsets. *J. Log. Algebraic Methods Program.*, 108:69–89, 2019.

18 Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. `doi:10.17487/RFC6749`.

19 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty session types for safe runtime adaptation in an actor language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2021.10`.

20 Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.

21 Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega Ojo, editors, *Distributed Computing and Internet Technology*, pages 55–75, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-19056-8_4`.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *POPL*, pages 273–284. ACM Press, 2008.

23 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. `doi:10.1145/2827695`.

24 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In Perdita Stevens and Andrzej Wąsowski, editors, *Fundamental Approaches to Software Engineering*, pages 401–418, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

25 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*, pages 116–133, Berlin, Heidelberg, 2017. Springer-Verlag. `doi:10.1007/978-3-662-54494-5_7`.

26 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

27 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2020.9`.

28 Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217`. Working Draft 17 December 2004.

29 Kerberos 5. `https://web.mit.edu/kerberos/krb5-1.19/`. Accessed: 14/02/2022.

30 Kickstarter. `https://www.kickstarter.com/about`. Accessed: 14/02/2022.

31 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP*, pages 146–159, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2967973.2968595`.

32 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types. In *36th European Conference on Object-Oriented Programming*, LIPIcs, 2022. in this volume.

33 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 221–232. ACM, 2015.

34 Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.

35 Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, 1992. `doi:10.1109/2.161279`.

36 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in typescript with routed multiparty session types. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, pages 94–106, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3446804.3446854`.

37 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: Compile-time API generation of distributed protocols with refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 128–138, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3178372.3179495`.

38 Rumyana Neykova and Nobuko Yoshida. *Featherweight Scribble*, volume 11665 of *LNCS*, pages 236–259. Springer, Cham, 2019. `doi:10.1007/978-3-030-21485-2_14`.

39 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local verification of global protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

40 Object Management Group. Business Process Model and Notation. `http://www.bpmn.org`.

41 Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*, pages 973–982, 2007. `doi:10.1145/1242572.1242704`.

42 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2011.

43 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECOOP.2017.24`.

44 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *PACMPL*, 3(POPL):30:1–30:29, 2019.

45 Paula Severi and Mariangiola Dezani-Ciancaglini. Observational equivalence for multiparty sessions. *Fundamenta Informaticae*, 170:267–305, 2019. URL: `http://www.di.unito.it/~dezani/papers/sd19.pdf`.

46 Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies. *J. Log. Algebr. Meth. Program.*, 95:17–40, 2018.

47 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. `doi:10.1145/3485501`.

48 Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In Evripidis Bampis and Aris Pagourtzis, editors, *Fundamentals of Computation Theory*, pages 18–35, Cham, 2021. Springer International Publishing.

49 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. In *OOPSLA 2020: Conference on Object-Oriented Programming Systems, Languages and Applications*, number OOPSLA (Article 148) in PACMPL, page 30 pages, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3428216`.

A Deterministic Memory Allocator for Dynamic Symbolic Execution

Daniel Schemmel
Imperial College London, UK

Julian Büning
RWTH Aachen University, Germany

Frank Busse
Imperial College London, UK

Martin Nowack
Imperial College London, UK

Cristian Cadar
Imperial College London, UK

Abstract

Dynamic symbolic execution (DSE) has established itself as an effective testing and analysis technique. While the memory model in DSE has attracted significant attention, the memory allocator has been largely ignored, despite its significant influence on DSE.

In this paper, we discuss the different ways in which the memory allocator can influence DSE and the main design principles that a memory allocator for DSE needs to follow: support for external calls, cross-run and cross-path determinism, spatially and temporally distanced allocations, and stability. We then present KDAlloc, a deterministic allocator for DSE that is guided by these six design principles.

We implement KDAlloc in KLEE, a popular DSE engine, and first show that it is competitive with KLEE's default allocator in terms of performance and memory overhead, and in fact significantly improves performance in several cases. We then highlight its benefits for use-after-free error detection and two distinct DSE-based techniques: MoKlee, a system for saving DSE runs to disk and later (partially) restoring them, and SymLive, a system for finding infinite-loop bugs.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases memory allocation, dynamic symbolic execution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.9

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.13

Funding This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement no. 819141 and 966733).

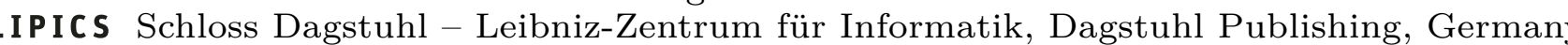
Acknowledgements We would like to thank Jordy Ruiz and the anonymous reviewers for their valuable feedback on the paper.

1 Introduction

Dynamic symbolic execution (DSE) [11] is a software testing technique that relies on systematically exploring the execution paths that a program might take, using a constraint solver to reason about the feasibility of each path.

An important component of a DSE engine is its memory model, which has received significant attention from the research community [5, 10, 16, 24, 32]. However, one component of the memory model has been largely ignored: the memory allocator. But the memory

© Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar;
licensed under Creative Commons License CC-BY 4.0



allocator plays a key role in DSE, with a direct influence on its ability to find memory-safety bugs and ensure deterministic execution, which is needed by several DSE-based techniques. Furthermore, the memory allocator can impact the performance and memory consumption of DSE, particularly in the EGT [9] style of DSE, where multiple program paths are concurrently kept in memory.

The memory allocator influences DSE's ability to find memory-safety violations because such errors can sometimes be detected only under certain memory layouts. For instance, a DSE engine like KLEE [8] may miss a buffer overflow if the pointer overflows into another memory object or may miss a use-after-free error if another object is allocated at the same location before a freed pointer is incorrectly dereferenced. In both cases, the final read or write appears to target valid memory, causing the engine to overlook the error.

The memory allocator influences deterministic execution in two ways: across runs and across paths. Most obviously, a non-deterministic allocator may impact determinism across two otherwise identical DSE runs (*cross-run determinism*). Less obviously, the memory allocator may also impact determinism across paths if the allocation decisions on one path influence the allocation decisions on another path (*cross-path determinism*). Determinism is important in DSE in many different scenarios. For instance, it makes it possible to compare multiple DSE configurations, such as using different constraint solvers [22] or search heuristics [6]. It also facilitates debugging when one needs to execute the same path repeatedly. Determinism is also important for scenarios that rely on re-runs, such as saving a run to disk and later restoring it [7], or re-executing events in the context of partial-order reduction for multi-threaded code [27]. Finally, determinism is also key for scenarios that compare memory contents across paths, such as finding infinite loops [26] and pruning redundant paths [4].

In this paper, we present KDAlloc, a memory allocator specifically targeting DSE. KDAlloc is carefully designed to achieve cross-run and cross-path determinism, maximise the probability of finding memory-safety bugs, keep a low memory and performance overhead, and allow the interaction with the outside environment. (The latter is an important distinguishing characteristic of DSE, and refers to its ability to interact with uninstrumented/unavailable code, such as external libraries and the operating system.)

We implement KDAlloc in KLEE [8], a popular DSE system based on the EGT [9] approach of keeping multiple paths concurrently in memory. We first show that KDAlloc is generally competitive with KLEE's default allocator in terms of performance and memory overhead, and that using it sometimes leads to significant performance gains. We then highlight its benefits for use-after-free error detection and in improving two distinct DSE-based techniques: MoKlee [7], a system for saving DSE runs to disk and later (partially) restoring them, and SymLive [26], a system for finding infinite-loop bugs.

In summary, the main contributions of this paper are:

1. An investigation of the main properties desirable from a memory allocator in a DSE context.
2. The design of KDAlloc, a new memory allocator for DSE, and its implementation into KLEE, a popular DSE system. KDAlloc is available as open source, together with an associated artifact.
3. An evaluation of KDAlloc in terms of performance and memory overhead, and its effectiveness on three scenarios: detection of use after free, MoKlee and SymLive.

The rest of the paper is structured as follows. §2 introduces some background information concerning DSE and its interaction with the memory allocator. §3 defines desired properties of an allocator for DSE engines. §4 presents the high-level design of KDAlloc, while §5

discusses several implementation details. §6 presents our experimental evaluation, on a diverse set of 18 benchmark suites with a total of 94 applications; and on two case studies: MoKlee and SymLive. Finally, §7 discusses related work and §8 concludes.

2 Background

Dynamic symbolic execution (DSE) executes programs on *symbolic inputs*. On each explored path, DSE maintains a *symbolic store* mapping variables to symbolic expressions and a *path condition (PC)* which describes the inputs following that path. For example, the symbolic store might map a program variable `x` to the symbolic expression $\lambda + 1$, while the symbolic input λ is constrained by the PC such that $\lambda \geq 0 \wedge \lambda < 100$.

Whenever execution reaches a branch point that depends on the symbolic input (e.g., `if (x == 0)` with $\texttt{x} \mapsto \lambda + 1$), the path is forked into two new paths: one following the `then` side, where the branch condition is added as a conjunct to the PC ($\text{PC}' = \text{PC} \wedge \lambda + 1 = 0$), and one following the `else` side, where the negation of the branch condition is added as a conjunct to the PC ($\text{PC}'' = \text{PC} \wedge \lambda + 1 \neq 0$). If either of these new PCs is unsatisfiable, execution does not continue along that path, as no concrete input exists which would cause the program under test to take that path through the program. In the example, `x` can never be zero. The symbolic store maps `x` to the symbolic expression $\lambda + 1$, which is constrained to be in the interval $[0, 100)$, meaning that $\lambda + 1$ has to be at least 1. Therefore, the `then` case (with $\text{PC}' = \lambda \geq 0 \wedge \lambda < 100 \wedge \lambda + 1 = 0$) cannot be triggered.

In the EGT [9] variant of DSE, all paths under exploration are kept in memory as *symbolic states*. Each symbolic state stores all the information necessary to continue execution on that path. In particular, each symbolic state has its own address space: globals, stack and heap.

One distinguishing characteristic of DSE is its ability to interact with the outside environment, such as external libraries and the operating system. In order to be able to perform such an *external function call*, a state needs to share its address space with the address space of the external library. This imposes an important limitation on the way memory is managed by the DSE engine, and thus on the memory allocator. For instance, KLEE manages this by having all states allocate memory in the unique address space of the KLEE process. Note that while two states with the same parent state may both have an object allocated at the same address, the object contents are unique to each state and stored in separate, internal memory buffers. However, before an external function call, the object contents are copied to their assigned allocation address in the address space of the KLEE process, so that external functions can work as expected. Similarly, after the external call completes, any changes made by the external code are propagated back from the KLEE address space to the internal memory buffers associated with the current state.

3 Design Principles

Traditional memory allocators [2, 13, 17, 18] are primarily concerned with performance and memory consumption. To achieve this, allocators try to keep the overhead of the allocator's operations low and to reduce memory fragmentation. However, allocator performance and memory consumption are not the primary considerations in a DSE context, as other operations overshadow them. Instead, we identify six key principles that need to guide the design of a memory allocator for DSE:

3.1 Support for External Calls

As discussed in §2, the ability to interact with the external environment is one of the main strengths of DSE. In order to be able to perform an external function call, a state needs to share its address space with the address space of the external function. Therefore, a memory allocator for DSE needs to manage not only the virtual address spaces associated with each state, but also the global address space which is used by external code.

3.2 Cross-run Determinism

One important property of a DSE engine is to have multiple identical runs behave in the same way. As discussed in the introduction, this makes it possible to compare multiple DSE configurations, e.g. with different constraint solvers [22]; facilitates debugging; and enables applications that rely on re-runs, such as saving a run to disk and later restoring it [7].

One reason for which different runs may behave differently is that program behaviour can depend on the memory layout. One simple example is the `memmove` function further described in §6.5, which changes behaviour depending on the relative locations of the source and destination addresses.

To remove this source of non-determinism, a DSE engine needs to use a cross-run deterministic memory allocator.

▶ **Definition 1** (Cross-run Determinism). *A DSE engine or memory allocator is* cross-run deterministic *iff its behaviour is the same for each run that is initialised in the same way.*

For instance, KLEE provides a simple deterministic memory allocator which internally allocates (via `mmap`) a large memory region at a fixed address, and then serves allocations from this region, never freeing objects. KLEE's deterministic allocator is cross-run deterministic, but it is often unusable in practice since it never frees memory.

Of course, a cross-run deterministic memory allocator does not suffice to have a cross-run deterministic DSE engine, as the latter may have other types of non-determinism (e.g., through the interaction with the environment).

3.3 Cross-path Determinism

As discussed in §2, the EGT style of symbolic execution stores multiple symbolic states in memory. If these symbolic states were to influence one another, the result of the analysis could suddenly depend on the order in which symbolic states are analysed. This is undesirable, as it makes it difficult to re-execute individual paths in isolation, which is needed in e.g. a debugging context or for selectively re-executing program paths.

▶ **Definition 2** (Cross-path Determinism). *A DSE engine is* cross-path deterministic *iff the behaviour of one symbolic state does not impact the behaviour of another.*

For a DSE engine to be cross-path deterministic, the memory of each symbolic state must be managed independently. Otherwise, any allocation would impact the shared memory allocator state and might change the memory allocation pattern of another symbolic state.

For instance, KLEE provides two allocators: the default one which simply uses the underlying system allocator, and the deterministic allocator described above. Since the allocator state is shared across states, none of them is cross-path deterministic, although the latter is cross-run deterministic.

By contrast, the memory allocator in EXE [10] is cross-path deterministic, because symbolic forking in EXE uses the UNIX system call `fork`, which duplicates the allocator's state.

3.4 Spatially Distanced Allocations

As noted above, traditional memory allocators aim to reduce memory fragmentation, in order to decrease the working set of a program and improve cache locality. On the other hand, by performing compact allocations, they also make it more probable that out-of-bounds accesses point to other valid objects.

For DSE, the latter is a much more important consideration. First, and as argued before, performance is a secondary aspect given the other large overheads of DSE. Second, addresses generated by the allocator are only actually used for the duration of an external function call, so the effect of the increased fragmentation becomes much less pronounced.

By contrast, finding buffer overflows is an important application of DSE. As most common out-of-bounds errors result in accesses that are in close proximity to their target object (e.g., off-by-one array indices), accessing them should not result in a valid access to a different object. Instead, allocations should be separated as far as possible to enable the detection of such overflows.

Our benchmark results, which we believe to be typical for a 2 h run of KLEE, had up to 758 million allocations and up to 412 MiB live in the whole symbolic execution engine, with up to 134 MiB live in any symbolic state. In a 64-bit address space (even when considering the 48-bit physical address space that can actually be used), there is a lot of room to spatially distance those allocations.

We note that the DSE engine can implement other mechanisms to find buffer overflows, which do not depend on the memory layout. For instance, EXE [10] tracks referent objects for all pointers. However, in the context of DSE, such mechanisms are unfortunately fragile, because such tracking information is lost while executing external code.

3.5 Stability

A desired property of a memory allocator for DSE is to have allocations as stable as possible with respect to slight changes in the allocation pattern. For example, if two paths allocate the same objects except that one of them temporarily allocates an extra object, a stable allocator would give the same memory addresses to the common objects on the two paths.

Stability ensures that similar paths have similar memory layouts, improving the effectiveness of approaches that compare memory across states [26, 27]; and that similar paths generate similar queries, improving cache hit rates [8].

Unfortunately, stability is in direct conflict with the objective of having temporally distanced allocations, which we discuss next.

3.6 Temporally Distanced Allocations

While spatially distancing allocations is important for buffer-overflow detection, temporally distancing them is important for finding use-after-free errors.

At one extreme, KLEE's deterministic allocator never frees memory, and thus never reuses it, reliably detecting all use-after-free errors. At the other end, an allocator that eagerly reuses memory would miss most use-after-free errors when the DSE engine has no additional mechanisms for tracking referent objects.

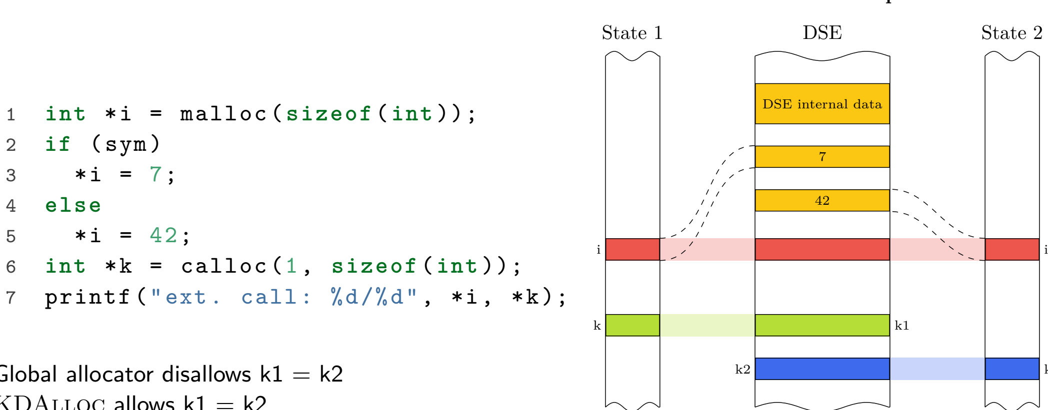


Figure 1 Code example with two paths illustrating how the state virtual address spaces are managed with a global memory allocator vs. KDAlloc.

As mentioned above, temporally distancing allocations is in direct conflict with the stability goal, as delaying memory reuse makes similar paths have different memory layouts. For example, if an object is allocated and freed again without any other memory allocation in between, it may be reused instantly iff allocations are not temporally distanced.

4 Design

Our allocator is guided by all six design principles discussed in §3. In this section, we first introduce the high-level design behind our allocator and discuss how it aligns with these design principles.

4.1 State Virtual Address Spaces

DSE engines like KLEE handle the allocated memory of all states via a global allocator. By contrast, KDAlloc is designed to manage the allocated memory for each state individually. This means the same virtual addresses can be used by different states, even for objects that are newly allocated by each state. This independence is a key element to achieve cross-path determinism and increase stability.

Figure 1 shows the difference between the way memory addresses are handled by a global allocator vs. KDAlloc. The program has two paths, encoded as two states by the DSE engine. State 1, shown on the left, runs the path that takes the `then` side of the branch, while State 2, shown on the right, runs the path that takes the `else` side.

Initially, there is a single state, State 1, which runs the `malloc` at Line 1. The DSE engine invokes the allocator and returns address i. When the symbolic condition at Line 2 is reached, another state, State 2, is forked. Immediately after the fork, both states share the same address space. When State 1 executes Line 3, it writes value 7 at the object allocated at address i. In a regular program execution, the address space (set of assigned addresses) and associated memory (the place where data is stored) are tightly connected, but these are handled separately in DSE. Therefore instead of placing value 7 at address i, State 1 places it into a separate memory location associated with State 1, as part of the DSE internal data. When State 2 executes Line 5, the write is handled in a similar way, by placing value 42 into a separate memory location associated with State 2.

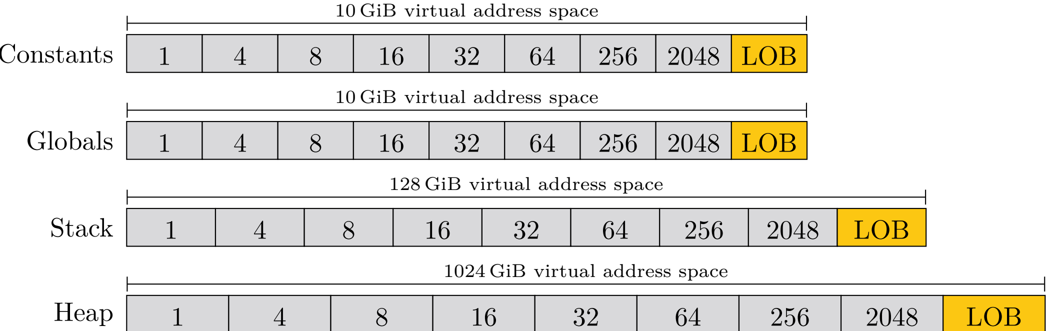


Figure 2 We decouple constants, globals, stack and heap by using different allocator instances. Each of these instances allocate virtual address space in the form of a large `mmap`ed area in main memory that is not backed by physical memory. (Virtual) sizes and base addresses for these are configurable with defaults suitable for common use. Each allocator instance is divided into 9 bins. The first 8 bins manage objects up to a maximum size (1–2048 bytes), while the last one, the large object bin (LOB), manages objects larger than 2048 bytes.

After these assignments, both states go on to execute the `calloc` at Line 6. It is here where the allocator makes a key difference. When a global allocator is being employed, as in KLEE, the two states will always receive *distinct* addresses, $k_1 \neq k_2$. By contrast, KDAlloc *can* return the identical addresses $k_1 = k_2$, as it manages addresses individually for each state. Furthermore, KDAlloc aims to return the same addresses when available, in order to increase stability.

When an external function call is invoked, as at Line 7, the assigned addresses need to be populated with the current set of values, so that external calls can find them in the expected place. In our example, the contents from the internal representation (in yellow) are copied to the concrete space at addresses i (for both states), k_1 (for State 1) and k_2 (for State 2).

In KDAlloc, the process is the same. The difference is that the concrete memory is backed by `mmap` and shared across the states, allowing the same addresses to be used between multiple states without them being related.

4.2 Memory Regions

For each state, KDAlloc maintains four memory regions: one for constants, one for globals, one for the stack and one for the heap. The reason for this separation is to increase stability: For instance, a change in the dynamic allocations should not impact stack allocations and vice versa.

4.3 Object Bins

We further divide these memory regions into bins for objects of a certain size: The first bin is for objects that have a single byte, the second for objects that are larger than 1 byte but not more than 4 bytes, the third one for objects that are larger than 4 bytes but not more than 8 bytes and so on up to objects of no more than 2048 bytes. A separate memory bin called the *Large Object Bin (LOB)* manages objects larger than 2048 bytes. The reason for having per-size bins is again to increase stability: In this way, a change in the allocation of objects within a certain size range does not impact the allocations for objects of other size ranges. In contrast, the reason for having a LOB is that for most programs only a minority of allocations are of large size.

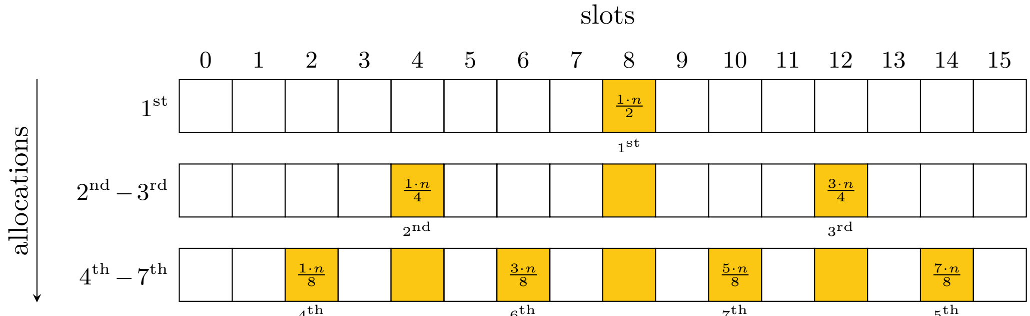


Figure 3 The slot allocator scatters object allocations throughout its bin to maximise inter-object distances. The figure shows a bin with $n = 16$ slots, and 7 allocations.

The bins managed by KDAlloc for each state are illustrated graphically in Figure 2. They are grouped into allocator instances (one each for constants, globals, stack, and heap). We determine the size of per-size bins by dividing the virtual address space assigned to its allocator instance by the total number of bins in the instance (i.e., 9) and rounding down to the next power of two. The LOB takes up the remaining virtual address space. The heap allocator instance, for example, defaults to a region that is subdivided into 8 per-size bins of 64 GiB each, while the LOB spans 512 GiB.

Each bin has its own allocator: the ones handling smaller objects use a *slot allocator* (§4.4), while the LOB uses a *large object allocator* (§4.5).

4.4 Slot Allocator

Each bin except for the LOB uses a slot allocator. A slot allocator divides the bin into n slots of equal size. For instance, a bin for objects of up to size 8 bytes is divided into n equal slots of 8 bytes each.

Typical slot allocators would allocate slots consecutively, to reduce fragmentation. As discussed in §3.4, a more important consideration in DSE is having spatially distanced allocations to prevent that out-of-bounds accesses point to valid objects. A common approach is to add fixed-size *redzones* [19,28] around allocations. Redzones are deliberately unallocated regions that trigger an error on any access. How far beyond the object bounds an access will always be detected as out-of-bounds depends on the redzone size.

As we are not held back by traditional fragmentation concerns, our algorithm aims to maximise the distance between allocations and thus the redzone sizes. The basic idea is to map the slots to a binary tree that is traversed level by level to identify the next free slot. Inside each level, nodes are traversed from the outside to the inside, to prevent the generation of long, monotonically increasing sequences of addresses. The root of this tree is in the middle slot. Every node divides the space of the parent node into two equally-sized spaces with the node itself marking the split.

Figure 3 illustrates this graphically. It depicts a bin with 16 slots, showing how slots are assigned over time. The first allocation is done in the middle slot, slot 8 (row 1 in the figure). Then, the second allocation is done in the middle of the left-side empty region, on slot 4, while the third allocation in the middle of the right-side empty region, on slot 12 (row 2). Finally, allocations 4 to 7 are done in the middle of the remaining regions, at slots 2, 14, 6 and 10 (row 3). With each level, the distance between objects becomes smaller.

On each allocation, this virtual binary tree is checked for slot availability in increasing level order, and the first available slot is always returned. Even if slots are allocated up to a level k, slots on lower levels may become available when objects are freed.

4.5 Large Object Allocator

For large allocations, a slot allocator is not ideal, as finding an appropriate slot size is difficult. If the size chosen is too small, large allocations will not fit, while if the size chosen is very large, most of the slot will usually be left unused.

To address this, our large object allocator takes another approach to manage the LOB. For every allocation, the largest free space is split equally, with the allocation placed at the split point. Similarly to the slot allocator, this algorithm keeps the distance between neighbouring objects as large as possible whenever an allocation occurs. On a deallocation, the freed space is merged with the regions to its left and right and returned to the allocator.

4.6 Quarantine

KDAlloc uses a quarantine [19, 28] to maximise the chance of finding use-after-free errors. That is, when an object is freed, instead of making the freed space available right away, a pointer to the object is instead placed in a quarantine region of a fixed size. When the quarantine becomes full, space is returned on a FIFO basis. This differs from other quarantine implementations in that the condition for releasing an object from the quarantine is not the total size of the quarantined objects, but rather just their number. We chose this method as the quarantine only governs the reuse of addresses and, unlike in quarantine implementations for general purpose allocators, quarantined objects do not take up any space beyond the metadata itself.

Each of the slot and large object allocators uses its own quarantine, to prevent deallocations in different bins from interacting with each other.

Choosing the quarantine sizes involves several trade-offs. Most obviously, larger quarantines take more space. On the other hand, larger quarantines can lead to the detection of more use-after-free errors (see §6.4), particularly those with a larger temporal distance between the deallocation and the invalid use. Quarantines also impact stability. For instance, consider again the example from §3.5, where two states allocate the same objects except that one of them allocates an extra object which is immediately freed. Without a quarantine, after the temporary allocation, the two paths will continue to have identical address spaces. However, with a quarantine the address spaces will start to differ, as the freed address cannot be reused immediately.

5 Implementation

We implemented KDAlloc as an alternative allocator in KLEE [8], a popular open-source DSE engine for LLVM bitcode.[1] Our implementation is written in C++ and has around 2K lines of code (excluding empty lines).

KDAlloc builds on the deterministic allocator from prior work on combining DSE with a partial-order reduction to symbolically analyse multi-threaded programs [27]. While that paper has only a superficial description of the deterministic allocator it uses ([27], §3.4), the allocator is made available as open source.[2]

[1] We use KLEE version 2.2.
[2] `https://github.com/por-se/por-se/blob/5786633/include/pseudoalloc/pseudoalloc.h`

KDAlloc reuses the basic structure of bitmap-based sized bins combined with a region-based large object bin, but significantly improves upon that allocator in many ways: Fork performance and memory usage are significantly improved by the addition of copy-on-write semantics, especially for the large object bin, which has been redesigned to use a singular treap with node-level copy-on-write (§5.5) instead of combining multiple `std::maps`. For the slot allocator, the scatter function has been improved to not generate long runs of monotonically increasing addresses (§4.4) due to visiting the implicit binary tree in level-order. We also completely redesigned how memory pages are returned to the OS after being used in external function calls (§5.3), as exploratory experiments showed the simple idea of always returning all pages to the OS immediately after an external function call leads to significant performance penalties in some cases.

5.1 Allocator Instances

The global and the constant memory regions are each associated with exactly one allocator that does not fork during the lifetime of KLEE, as the memory allocation of constants and globals does not change during the runtime of the program. We distinguish between globals and constants, because globals need to be written out on a per-state basis when an external function call occurs, while constants can be written only once. Note that, just like KLEE, we assume external calls are well-behaved, e.g., we do not implement any techniques to ensure that external function calls do not access memory outside their valid allocations. The heap and stack memory regions are used to initialise the heap and stack allocators when the initial state is created from scratch. After this has been done once, the allocators are forked when the state is forked. The allocator state can be forked efficiently by using a bin-level copy-on-write (CoW) mechanism. Initially, all bins are unallocated. Once an allocation happens, the respective bin is created and owned by the allocator. When a symbolic state is forked, so are the allocator states, which leads to both allocators having a shared CoW reference to each allocated bin. When an object is (de-)allocated in such a shared bin, the bin is copied and after that owned. If only one other allocator references the CoW bin, it regains exclusive access.

KDAlloc uses size information for deallocations, which KLEE provides as part of each memory object. It is not fundamentally necessary to use sized deallocations; if another symbolic execution engine does not store the size of the allocation in a similar manner, the address itself could be used to look up the appropriate bin, at which point a slot allocator knows the size of the allocation by default (1 slot) and the large object allocator can deduce the size by finding the empty spaces before and after the allocated object.

5.2 Virtual Memory Regions

The virtual memory regions used to back external function calls are created at user-provided base addresses. They are mapped as read and write, but non-executable, anonymous, private and without physical backing pages. Similarly, mainline KLEE's deterministic allocator allocates memory in much the same way, except not enforcing the non-backing of pages.[3]

The `fork` call is slow, especially when large amounts of memory are involved [1, 21]. To ensure that the impact of the allocator change on KLEE's usage of `fork` (to decouple solvers from the main process) is as small as possible, the memory region is marked as `MADV_DONTFORK`. This means that the region will not be available in child processes.

[3] `https://github.com/klee/klee/blob/7d85ee8/lib/Core/MemoryManager.cpp#L70-L71`

5.3 External Function Calls

Mainline KLEE creates an initial buffer when a new memory object is created by the program under test, which is used to acquire a memory address that can be used for future external function calls (i.e., calls to uninterpreted functions). When KDAlloc performs an external function call, it copies the data out to the virtual memory region instead. Since the allocator performs allocations with maximal distance, it will usually not share memory pages between allocations. This means, that a one byte allocation will probably require a full page (usually 4096 B) for the duration of the external function call. After the external function call has been performed, any changes are copied back into the symbolic state, and the physical pages are not needed anymore.

To reduce memory consumption, we inform the OS that it may reclaim the physical pages at its leisure by using `madvise` with `MADV_DONTNEED`. In Linux, this operation is, however, relatively costly as its runtime depends on the size of the mapping, not just the number of active pages. As many external function calls only require a comparatively small number of pages, we run `madvise` only once the number of active pages exceeds twice the average number of pages needed for a single external function call (with a minimum of 1024 pages/4 MiB). To compute the average of pages needed for one external call, we use an exponential moving average: $\text{avg}_{i+1} = \frac{3 \cdot \text{avg}_i}{4} + \frac{\text{call}_i}{4}$. We approximate the number of pages needed for a single external function call based on the number of involved objects, by assuming that no two objects share a memory page, which is often the case as we maximise inter-object spacing. The total number of pages can be easily determined by using `getrusage`. We exclude allocations in the global constants mapping from this mechanism, as they do not change in between external function calls.

5.4 The Slot Allocator

The slot allocator is used for allocations smaller or equal to 2048 B (§4.4). It uses two different methods of naming a slot, the *position* and the *index*. The position is the actual address of the allocation relative to the base address of the associated virtual memory region. The index reflects the node position while traversing the binary tree (§4.4) and is used as an index into a bitmap that stores whether the slot is currently allocated or not.

Spreading out allocated objects maximises the distances between allocations. When a maximum of n allocations were live at the same time (including those in quarantine), the distance between any two allocations and between the beginning or end of the bin and any allocation is at least $\mathit{size}/2^{\lceil \log_2(n+1) \rceil} - \mathit{slotsize}$. The slot allocator asserts that the distance will always be greater than zero, which ensures that no two directly adjacent slots will be used, which in turn ensures that a slot-allocator administrating slots of size s will always leave at least s bytes in between any two allocations and to the allocator using the region before it (slot allocators are assigned ascending regions, so it will also have at least s bytes unused after the last allocated slot).

While it may seem like this method wastes a lot of memory to increase robustness, this is only the case when viewed from the program under test and for the duration of external function calls, when objects of a single state are copied into the virtual memory regions. By transforming between position and index, the bitmap is kept compact while the managed objects are spread out.

5.5 The Large Object Allocator

As explained in §4.5, the large object allocator manages a number of free regions and performs allocations in the middle of the largest such free region, splitting it into two. It manages memory in blocks of 4096 B and ensures that at least one such 4096-B block remains unused in between any two allocated objects.

Since objects can be of very different sizes, the large object allocator does not utilise a simple bitmap, but rather uses a treap, i.e., a data structure that is both a search tree and a max heap, to store free regions. The addresses of the free regions are used as keys in the search tree and their sizes are used as priorities to organise the max heap. This enables fast lookups of regions by their address as well as quick identification of the largest one. However, as there may be multiple equally-sized regions, and therefore multiple regions that contend for being the largest one, we also use a perfect hash of the addresses as secondary priorities. Thus, every priority is unique, and the treap assumes a unique shape.

To allocate new memory, the root of the treap (which is also the top of the heap) is removed. The object is situated in the middle of that free region, and the newly generated redzones are reinserted into the treap. If the object is larger than the largest free region, or not large enough to retain redzones of at least 4 KiB before and after, the allocation fails.

To free an allocation, the address of the allocation is used to find the free regions immediately before and after the object to be freed. These regions are removed from the treap and combined back into one region, covering both redzones and the object, before being reinserted.

To save memory, each node of the treap is shared using reference-counted copy-on-write. Since each node contains references to its children, this means that sub-treaps can be shared between multiple allocator states.

5.6 Quarantine

The quarantine is a per-bin FIFO queue implemented as a ring buffer that only allocates when it is needed (i.e., only when the quarantine is neither zero nor infinite) and in use. Each bin uses its own quarantine queue to prevent deallocations in different bins from flushing previous deallocations in otherwise unrelated bins. This trade-off comes at a slight increase in memory overhead, as each bin needs a k-element quarantine buffer to give a global guarantee that at least the last k deallocations are being delayed by the quarantine.

In §4.6, we discussed the various trade-offs involved in choosing the quarantine sizes. Based on initial experiments, we have decided on a default value of 8 slots per quarantine, but kept the size configurable.

6 Evaluation

In this section, we evaluate the overheads and benefits of KDAlloc. After presenting the experimental setup (§6.1), we measure the memory overhead and performance impact of KDAlloc by comparing it to the default allocator of mainline KLEE (§6.2) and explore the root cause of the solver time improvements that were observed in some of the benchmarks (§6.3). We then evaluate the benefits KDAlloc provides in the context of use-after-free error detection (§6.4) and two KLEE-based projects: MoKlee (§6.5) and SymLive (§6.6).

Table 1 KLEE benchmarks.

Suite	Version	Apps
GNU awk	5.1.0	`awk`
GNU bc	1.07.1	`bc`
GNU Binutils	2.35.1	9 tools[4]
GNU Coreutils	8.32	68 tools
GNU datamash	1.7	`datamash`
GNU Diffutils	3.7	`diff`
GNU Findutils	4.7.0	`find`
Libtasn1	4.16.0	`asn1Decoding`
libTIFF	4.1.0	`tiffdump`, driver[5]
libxml2	2.9.10	driver[6]
GNU M4	1.4.18	`m4`
GNU Make	4.3	`make`
ImageMagick	7.0.10-45	`magick`
oSIP	5.2.0	driver[5]
GNU sed	4.8	`sed`
SQLite	3340000	`sqlite3`
tcpdump	4.9.3	`tcpdump`
Vorbis Tools	1.4.0	`oggenc`

Table 2 SymLive benchmarks.

Suite	Version	Apps
BusyBox	1.27.2	`hush`, `sed`, `yes`
GNU Coreutils	8.25	`ptx`, `tail`, `yes`
GNU regex	0.12	driver
GNU sed	4.4	`sed`
Toybox	0.7.5	`sed`, `yes`

Table 3 MoKlee benchmarks.

Suite	Version	Apps
GNU Binutils	2.33	`readelf`
GNU Coreutils	8.31	87 tools
GNU Diffutils	3.7	`diff`
GNU Findutils	4.7.0	`find`
GNU Grep	3.3	`grep`
libspng	`#2079ef6`	driver
tcpdump	4.9.3	`tcpdump`

6.1 Experimental Setup

To minimise the non-deterministic impact of the environment, we ran all experiments on a cluster of homogeneous machines (Intel Core i7-4790 @ 3.6 GHz with 16 GiB RAM) with an equivalent OS/library setup (Ubuntu 18.04). Each experiment is executed inside a Docker container that allows reproducing a similar execution setup on each machine. For mainline KLEE and SymLive we use LLVM 11.0 and Z3 4.8.8, whereas MoKlee is linked against LLVM 3.8 and uses Z3 4.8.4.

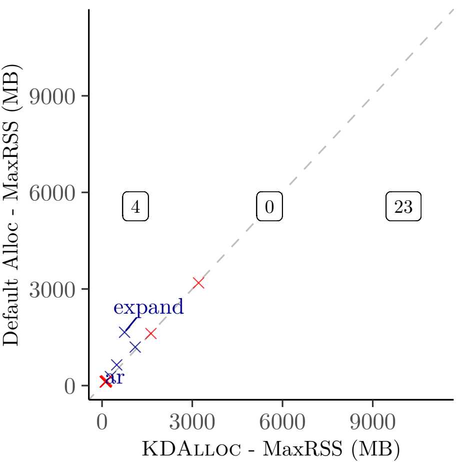
All of our experiments are available at `https://doi.org/10.5281/zenodo.6540857`.

6.2 Memory Consumption and Performance

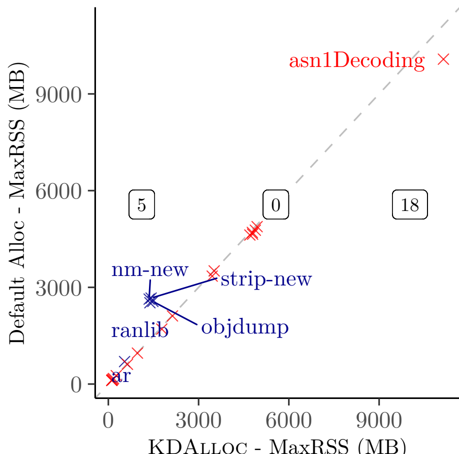
We implemented KDAlloc as an additional allocator in mainline KLEE 2.2 and tested it against a diverse set of 18 benchmark suites ranging from basic system utilities to databases or image processing tools. From each suite, we chose the most representative applications and test drivers made available in recent KLEE-related publications. In total, we tested 94 applications (Table 1). However, we only used a subset of the GNU Coreutils applications: We excluded all applications that interfere with our test setup (e.g. `chmod`, `truncate`), crash KLEE due to a known bug in the Z3 front-end (e.g. `ptx`), are very similar or aliases to other tools (e.g. `dir`), or are not compatible with our deterministic thresholds described below (e.g. `fmt`).

Comparing two alternative implementations is a non-trivial endeavour, especially with EGT-style symbolic execution engines. Every change in exploration, memory allocation, environment, state termination due to memory pressure etc. can cause completely different executions and invalidate the comparison. Hence, it is of utmost importance to eliminate

[4] Binutils: `addr2line`, `ar`, `elfedit`, `nm`, `objdump`, `ranlib`, `readelf`, `size`, `strip`
[5] Driver from: `https://figshare.com/articles/code/ESEC_FSE_2020_PSPA_artifact/12410231`
[6] Driver from: `https://github.com/davidtr1037/klee-mm-benchmarks`



(a) DFS – 4 points above the diagonal, 23 points below the diagonal and none on the diagonal.



(b) RndCov – 5 points above the diagonal, 18 points below the diagonal and none on the diagonal.

Figure 4 MaxRSS for KDAlloc vs the default allocator using different searchers for runs that are deterministic. Points with relative difference of at least 10% are labelled. Points above the diagonal are blue and points below the diagonal are red. The number of points above, below and on the diagonals are noted in each graph.

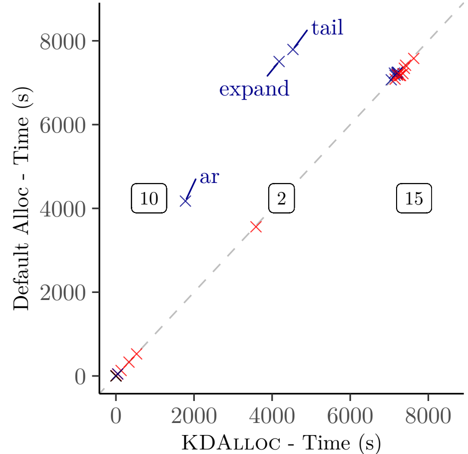
most sources of non-determinism. We tried to mitigate the issue by slightly modifying KLEE: First, we replaced all timer-based thresholds by instruction-based thresholds. Second, we similarly replaced memory-based thresholds by thresholds based on the number of states. Third, we added more statistics, such as the number of allocations or external function calls, to evaluate and compare executions.

Initially, we ran each application for 2 h with the default 2 GB memory limit, unmodified thresholds, and the default allocator. From these runs we derived suitable values for the more deterministic thresholds for each application. Instead of running experiments for 2 h with a 2 GB memory limit and logging intervals of e.g. 30 s, we can now use much more precise descriptions and run an application for n instructions, re-compute coverage every x instructions, update logs every y instructions and terminate states when we reach a maximum of z states.

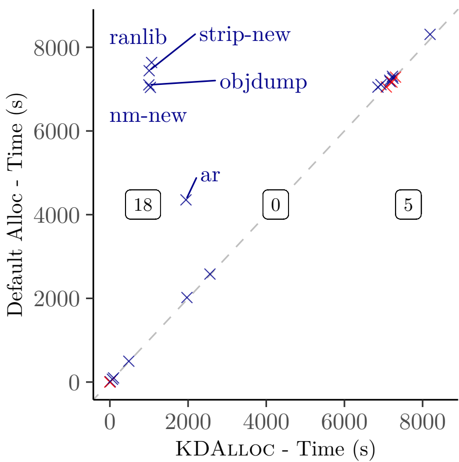
For a few applications such as `fmt`, we were not able to find a working threshold for the maximum number of active states, as KLEE's performance for these applications degrades from 100k active states to 1-10 states over time. A low threshold leads to an immediate termination of KLEE whereas a high threshold causes memory exhaustion before the targeted run time.

After acquiring the thresholds for KLEE's default exploration strategy (RndCov), a combination of random-path traversal and a distance-based approach to target uncovered code [8], we repeated the process for the simpler depth-first search strategy (DFS).

With these more deterministic configurations, we ran each experiment with both exploration strategies (DFS/RndCov) and both allocators five times. We consider an application deterministic (or *comparable*) under a search strategy when all ten runs, five for each allocator, show the same values for core statistics, in particular the same number of instructions, covered instructions, allocations, queries, and external calls.



(a) DFS – 10 points above the diagonal, 15 points below the diagonal and 2 on the diagonal.



(b) RndCov – 18 points above the diagonal, 5 points below the diagonal and none on the diagonal.

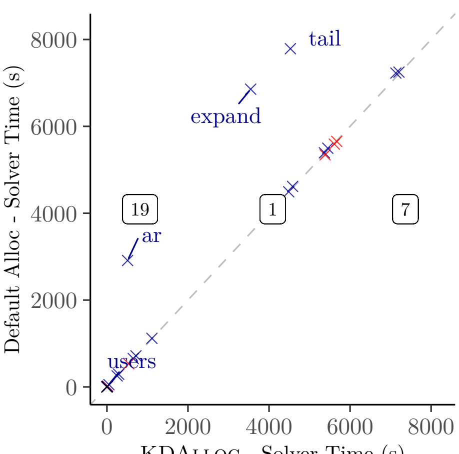
Figure 5 Execution time for KDAlloc vs the default allocator using different searchers for runs that are deterministic. Points with relative difference of at least 10% are labelled. Points above the diagonal are blue and points below the diagonal are red. The number of points above, below and on the diagonals are noted in each graph.

To understand the memory consumption of KDAlloc, we recorded the maximum resident set size (MaxRSS) for all runs, as reported by the Linux kernel. As seen in Figure 4a for DFS and Figure 4b for RndCov, the two allocators generally use the same amount of memory. There are several outliers where the differences are slightly larger (but still small): For most of them KDAlloc consumes less memory, but there is also one case (`asn1Decoding` for RndCov) where KDAlloc consumes more. The reason KDAlloc can consume more memory is through its more expensive metadata when many small allocations are involved. On the other hand, it can consume less memory since it only uses the buffers corresponding to the addresses it returns when an external call is performed (in contrast, the default allocator uses `malloc` to reserve that space).

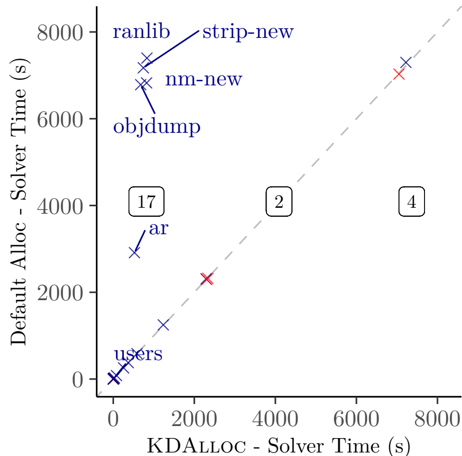
To better understand the nature of our benchmarks, we measured several statistics across all benchmarks and both search heuristics. KDAlloc had to handle up to 758M allocations (median 11M), allocate up to 5157 MiB (median 97 MiB), and manage up to 5.5M live allocations (median 9154) and 412 MiB (median 625 KiB) of live memory in a single experiment run. A single state during symbolic execution could perform up to 758M allocations (`yes` with DFS) (median 2706), allocate up to 4896 MiB (median 57 KiB), and keep 7899 allocations (median 640) or 134 MiB (median 16 KiB) live.

In Figure 5a (DFS) and Figure 5b (RndCov) we compare the execution time when KDAlloc is used (x-axis) with the execution time when the default allocator is used (y-axis). For most of the tested applications, the execution time is similar. This shows that KDAlloc does not impose any significant performance overhead. In contrast, all outliers (with a relative difference of at least 10%) are above the diagonal indicating that KDAlloc can lead to significant speedups for certain benchmarks.

To explain the potential cause of those positive outliers, we analysed the solving time, shown in Figure 6a for DFS and Figure 6b for RndCov. The graphs confirm that the main reason for the speedup achieved by KDAlloc for those benchmarks is the time spent solving constraints.



(a) DFS – 19 points above the diagonal, 7 points below the diagonal and 1 on the diagonal.



(b) RndCov – 17 points above the diagonal, 4 points below the diagonal and 2 on the diagonal.

Figure 6 Solver time using different searchers for runs that are deterministic with and without KDAlloc. Points with relative difference of at least 10% are labelled. Points above the diagonal are blue, points below the diagonal red and points exactly on the diagonal are black. The number of points above, below and on the diagonals are noted in each graph.

6.3 Solver Time Improvements

At the beginning, we suspected a higher cache-hit rate as the main reason for KDAlloc speeding up constraint solving. Due to higher stability (§3.5), KDAlloc leads to more hits in KLEE's caches (the query cache and the counterexample cache [8, 22]). If query expressions share common addresses, we may have more identical queries and solutions are also more likely to be reused, which can have a great impact on caching efficacy [20, 31]. Indeed, the outlier benchmarks show an increased number of hits. However, the relative difference is small, at just over 0.01% of all queries, and did not explain the overall speedup.

Therefore, we hypothesised that the different addresses returned by KDAlloc may make a large number of queries easier to solve. To investigate this, we tracked all the solver calls for KDAlloc and the default allocator and compared their execution time for the `expand` benchmark (DFS).

In both configurations, KLEE issued a total of 1,274,757 queries. Of these, 758,079 queries were different, depending on the allocator. We could match all but 2604 of those queries by mapping the constants that relate to the non-deterministic addresses returned by the default allocator to the corresponding constants that relate to the deterministic addresses returned by KDAlloc. The remaining differences were minor and due to sources of non-determinism that were neither addressed by our work nor had an impact on the deterministic execution of `expand`.

While for each individual query the absolute time difference is only a few milliseconds, many of the queries with deterministic addresses can be solved roughly twice as fast as their non-deterministic counterpart. With such a large number of queries, these differences add up and explain the overall solver speedup.

We manually inspected the 100 queries that show the most significant absolute time improvement. These queries involve bounds checks and constraints on single bytes of a symbolic file (originating from `expand`'s main loop) and show high similarity. For these

```
(Extract w32 0
    (Add w64 0xFFFFDDBC00000000 (Select w64 C 0x0000000000000000 0x0000224400000000)))
——— ↓ Extract(Add): (Extract (Add x y)) → (Add (Extract x) (Extract y)) ———
(Add w32 (Extract w32 0 0xFFFFDDBC00000000)
    (Extract w32 0 (Select w64 C 0x0000000000000000 0x0000224400000000)))
——— (Extract w32 0 0xFFFFDDBC00000000) → 0x00000000 and ↓ Extract(Select) ———
(Add w32 0x00000000
    (Select w32 C (Extract w32 0 0x0000000000000000) (Extract w32 0 0x0000224400000000)))
——— (Extract w32 0 0x0000000000000000) → 0x00000000 ———
——— (Extract w32 0 0x0000224400000000) → 0x00000000 ———
(Add w32 0x00000000 (Select w32 C 0x00000000 0x00000000))
——— (Select w32 C 0x00000000 0x00000000) → 0x00000000 ———
(Add w32 0x00000000 0x00000000) = 0x00000000
```

(a) KDAlloc's address structure can enable offset reasoning independent of base addresses.

```
(Extract w32 0
    (Add w64 0xFFFFAAAAA7290C00 (Select w64 C 0x0000000000000000 0x0000555558D6F400)))
——— ↓ Extract(Add): (Extract (Add x y)) → (Add (Extract x) (Extract y)) ———
(Add w32 (Extract w32 0 0xFFFFAAAAA7290C00)
    (Extract w32 0 (Select w64 C 0x0000000000000000 0x0000555558D6F400)))
——— (Extract w32 0 0xFFFFAAAAA7290C00) → 0xA7290C00 and ↓ Extract(Select) ———
(Add w32 0xA7290C00
    (Select w32 C (Extract w32 0 0x0000000000000000) (Extract w32 0 0x0000555558D6F400)))
——— (Extract w32 0 0x0000000000000000) → 0x00000000 ———
——— (Extract w32 0 0x0000555558D6F400) → 0x58D6F400 ———
(Add w32 0xA7290C00 (Select w32 C 0x00000000 0x58D6F400))
```

(b) The same simplification is not possible with address-dependent constants from the default allocator.

Figure 7 Example of query simplification enabled by KDAlloc. Steps performed by Z3's pre-processing stage are shown in red. The highlighted simplification is only observed with KDAlloc.

queries, we found that the time differences can be explained by the initial pre-processing stage of Z3 [12], which was able to greatly simplify the queries generated when using KDAlloc before invoking the core solver.

The top of Figure 7a shows a fragment of one of the queries that depends on the allocator used and where the address-dependent constants, `0xFFFFDDBC00000000` and `0x224400000000`, were obtained using KDAlloc. The top of Figure 7b shows the same fragment, but with the address-dependent constants stemming from the default allocator: `0xFFFFAAAAA7290C00` and `0x555558D6F400`. The queries are given in the KQuery format,[7] where `(Select [width] [condition] [true-expr] [false-expr])` is an *if-then-else* operation that evaluates to either `true-expr` or `false-expr` (both of the same given `width`) depending on whether `condition` evaluates to *true* or *false*. The `(Extract [width] [index] [expr])` operation evaluates to the `width` least-significant bits taken from `expr`, omitting the first `index` bits. In both fragments, we omit a complex expression denoted by `C`.

In these fragments, only the lower 32 bits of the result are significant (`Extract w32 0`). Using simple rewriting rules, the `Extract` operation is pushed down to the constants. In the KDAlloc case, this simplifies the expression enough to remove the `Select`, leaving a single constant. In the case of the default allocator, since the two operands that are to be chosen based on `C` are not the same, the `Select` cannot be removed.

[7] https://klee.github.io/docs/kquery/

Listing 1 Whereas KLEE with KDAlloc and a quarantine can detect the use after free at Line 9 reliably, KLEE with the default allocator can only detect it when the freed space is not reused in the meantime, e.g. by the second `strdup` or internal KLEE data structures.

```
1  char *mallocfree() {
2      char *s = strdup("A");
3      free(s);
4      char *t = strdup("B");
5      return s;
6  }
```

```
7  int main(void) {
8      char *s = mallocfree();
9      puts(s);
10     return 0;
11 }
```

Why do addresses issued by KDAlloc follow those specific patterns? First, the virtual address spaces are configured to start at addresses with all of the lower 32 bits set to zero. Next, the way we size our bins (see §4.3), along with the algorithm we use to assign slots (see §4.4), results in base addresses that are a sum of high powers of two for every object. As a result, all enquiries about the lower 32 bits of pointers can trivially be rewritten into reasoning about offsets, without involving the base address.

We also confirmed that STP [15] (another major solver used by KLEE) is able to perform a similar simplification during its initial pre-processing stage and shows similar improvements in solving time. We can thus conclude that the addresses returned by KDAlloc can have a positive effect on query solving time apart from caching.

6.4 Detection of Use-after-free Errors

KLEE can find many memory access violations. Its detection, however, is primarily focused on out-of-bounds accesses. To detect these, KLEE checks for every memory access whether the address does not point to a valid object, or in the case of a symbolic address whether it is possible not to point to a valid object. In such a case, an out-of-bounds error is reported.

If the program under test frees an allocation, its entry is removed from the list of valid objects in the current state. While many use-after-free (and the related double-free) errors can be found using this mechanism (i.e. the address does not resolve to a valid object), KLEE has been shown to sometimes miss even the simplest cases[8] or to depend critically on subtle aspects such as compilation flags[9] for their detection.

KLEE's use-after-free error detection is fragile as it depends on the freed space not being reused by a subsequent allocation. An example of this is shown in Listing 1. Without compiler optimisations enabled, KLEE usually fails to detect the use after free at Line 9, as the underlying default allocator usually reuses the address for the allocation of the string `"B"` or one of the local variables in the `strdup` or `puts` calls. With Line 4 removed or optimisations enabled, KLEE is able to find the bug.

KDAlloc, on the other hand, provides a quarantine for deallocated objects (§4.6). As long as objects remain quarantined, any use-after-free error is guaranteed to be detected. In addition, KDAlloc greatly enhances the comprehensibility of reported use-after-free errors, which in baseline KLEE are reported as out-of-bounds accesses. If an address is not resolved to a valid object, but the location is still marked as allocated in the allocator metadata, we conclude that it must be in quarantine, and thus it is a use-after-free error.

[8] https://www.mail-archive.com/klee-dev@imperial.ac.uk/msg02998.html
[9] https://github.com/klee/klee/issues/1434

Listing 2 An implementation of the ANSI C function `memmove` as found in the uClibc[10] library. The pointer values in the comparison at Line 4 depend on the values returned by the underlying allocator. The default allocator returned different values in the re-execution and caused a divergence.

```
1  void *memmove(void *dest, const void *src, size_t n) {
2    char *s = (char *) dest;
3    const char *p = (const char *) src;
4    if (p >= s) {
5      while (n) {
6        *s++ = *p++;
7        --n;
8      }
9    } else {
10     while (n) {
11       --n;
12       s[n] = p[n];
13     }
14   }
15
16   return dest;
17 }
```

Implementing a quarantine for the default allocator is possible [28], but would incur a significant space penalty for several reasons. First, only a small portion of the quarantined objects are actually objects from the program under test, as most are allocated by the DSE engine. Thus, the quarantine would have to be several times as large to provide similar benefits. Second, this would be a global quarantine, meaning that it would have to be even larger to provide similar per-state guarantees, even if the symbolic states are visited uniformly. Finally, such a quarantine causes memory fragmentation, as memory cannot be reused immediately; in the case of KDAlloc, this causes no additional memory pressure, as it only fragments the emulated address space of the program under test.

6.5 MoKlee

MoKlee [7] is an extension of KLEE implementing a variant of memoised symbolic execution [33]. MoKlee provides the ability to save an ongoing DSE run to disk and then to (partially) restore it back into memory via a fast replay process. More exactly, MoKlee saves to disk metadata, such as constraint solver results and path information, and re-uses this information during replay to remove the constraint solving cost. MoKlee can optionally filter out fully explored path subtrees, in a mode called *path pruning*.

This approach is applicable to real-world software as long as the engine is able to detect divergences [7]. Divergences occur when a DSE engine explores different code paths in different runs, although the controllable inputs are the same. This happens due to values that are read from the environment (e.g. date/time strings or disk usage) or, more relevant for our approach, when the execution relies on memory addresses. For instance, the branches taken to traverse a hash table with pointer values as keys depend on the addresses returned by the underlying allocator. If the allocator is non-deterministic and the allocation order changes in

[10] `https://www.uclibc.org/`

subsequent runs, the insertion order in the hash table changes and branch decisions cannot be re-used. KLEE's `mmap`-based deterministic allocator is shown to reduce divergences significantly, e.g. for `find`, but it is not suitable for most applications as it cannot free or re-use memory [7].

We ported KDAlloc to MoKlee, and added some statistics (e.g. number of external calls) and an instruction-based threshold for the coverage computation. Most of the other changes that were necessary for mainline KLEE could be mimicked by similar flags that were already available in MoKlee. As benchmarks we re-used the 93 applications (Table 3) provided with the MoKlee artifact.[11] The setup is similar to the mainline KLEE experiment: We start with 2 h memoisation runs using a 2 GB memory limit, as described in the MoKlee artifact, to find deterministic thresholds for the two exploration strategies (RndCov, DFS). After that, we re-run the memoisation pass for all applications with both exploration strategies and *both* allocators, but using deterministic thresholds. A total of 66 applications for DFS, respectively 38 applications for RndCov, had the same relevant statistics and hence executed the same paths with high probability across both allocators. In short, these runs are *comparable*.

To measure the influence of both allocators on divergences, we re-executed all memoised runs with and without path pruning using both exploration strategies as done in the MoKlee paper. We only omit the results for the non-pruning re-execution of memoised DFS runs with the RndCov search strategy, as the wide-and-deep shape of DFS execution trees often cause state explosions and hence state terminations in many benchmarks, making a comparison meaningless.

We refer to each experiment by the search strategies used during the memoised run and during replay. For instance DFS/RndCov denotes the experiment where the original memoised run used DFS, and the replayed run used RndCov.

Firstly, we evaluate the experiments based on comparable memoisation runs. Due to the nature of these tools, the number of divergences is low and we only observed them in three applications when re-executed without path pruning: `nohup` (RndCov/DFS) diverges in a comparison that checks the maximum number of open file descriptors, whereas `shred` (RndCov/DFS) diverges in a comparison of timestamps. Both applications diverge in exactly the same locations with both allocators and the root causes cannot be prevented by our allocator. However, `du` (DFS/DFS) only diverges with the default allocator. The reason for that is a pointer comparison in the implementation of the standard C function `memmove` (Listing 2) to copy bytes between memory areas. Memory areas are allowed to overlap, in contrast to `memcpy`, and most C libraries save a temporary buffer by comparing the source and destination pointers (Line 4). Depending on their order, the algorithm starts copying either from the front or the back of the areas to prevent overwriting the overlapping area too early. In our experiment, the default allocator returned different addresses for `p` and `s` during the re-execution, changing the order of both pointers in memory and causing a divergence that got detected by MoKlee. When a divergence occurs, MoKlee removes the memoised subtree and the affected subtree needs to be re-explored. In the case of `du`, a significant 32.5% of the memoised instructions were lost that way. With KDAlloc's deterministic allocation on the other hand, both pointers retrieved the same values during re-execution and the complete memoised run could be re-used.

[11] `https://zenodo.org/record/3895271`

Table 4 Number of source locations with diverging behaviour in applications with differing memoisation runs across allocators. Different numbers are observed between allocators for the first three benchmark suites.

	DFS		RndCov	
Suite	MoKlee	KDAlloc	MoKlee	KDAlloc
Coreutils	22	12	42	32
Findutils	1	0	1	1
Libspng	0	0	1	0
Binutils	0	0	0	0
Diffutils	0	0	0	0
Grep	0	0	1	1
Tcpdump	0	0	0	0

Secondly, we evaluate the remaining non-comparable applications where the memoisation runs between allocators differ. Here, we count the number of unique source locations where divergences occur across re-execution runs (with and without path pruning). As can be seen in Table 4, with KDAlloc divergences occur in significantly fewer locations for both search strategies, showing that KDAlloc is effective in preventing memory-related divergences.

Furthermore, in the original MoKlee paper `find` benefited most from a deterministic allocator. Our experiments confirm that observation. Starting from a RndCov memoisation run and re-executing that run without path pruning, KLEE has to terminate 3407 paths due to divergences with the RndCov exploration strategy (3545 for DFS) whereas KDAlloc does not observe a single divergence with RndCov and only 3 with DFS. In summary, KDAlloc reduces the number of diverging paths and hence improves the effectiveness of memoised symbolic execution.

6.6 SymLive

SymLive [26] is an open-source[12] KLEE extension that uses symbolic execution to find paths that lead to infinite loops. It hashes the symbolic states and checks for repeating hashes along each path. If such a repetition is found, the program transitions back into a previous program state: Assuming deterministic program execution, the program under test has entered an infinite loop, which SymLive reports if it violates a generic liveness property.

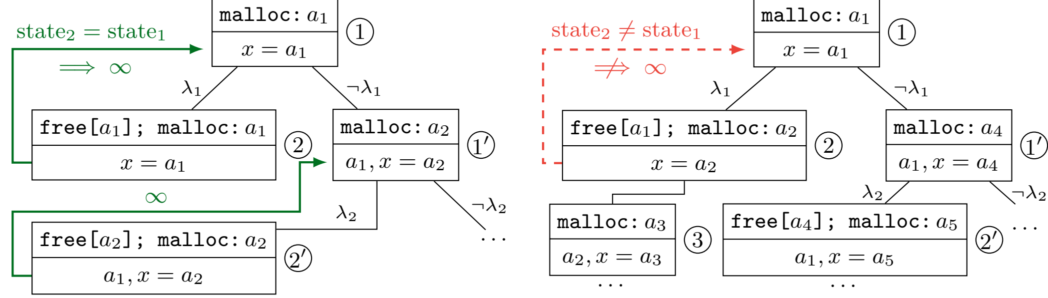
The original implementation of SymLive employs KLEE's default allocator, which is good enough in many practical applications [26]. The default allocator, however, is not cross-path deterministic (§3.3), which may prevent SymLive from detecting an infinite loop in one state if another state prevents address reuse.

The program in Listing 3 is one such example. It conditionally leaks allocations and prevents forked states from terminating. Its infinite behaviour can be reliably detected using KDAlloc with the quarantine disabled (the desired configuration in this context), but not using the default allocator. The example revolves around two pointers, `x` and `p`. The former is initially bound to a memory object, the latter is made symbolic (Line 2). The infinite loop (Lines 3–6) reassigns `x` with a freshly allocated memory object (Line 5). If `x == p`, the previous pointee is freed just before (Line 4); otherwise it is leaked.

[12] `https://github.com/COMSYS/SymbolicLivenessAnalysis`

Listing 3 Leaking memory objects in an infinite loop can hinder its detection in SymLive without cross-path deterministic allocation.

```
1 int main(void) {
2   void *x = malloc(1), *p = klee_symbolic_ptr();
3   while (1) {
4     if (x == p) free(x);
5     x = malloc(1);
6   }
7 }
```



(a) SymLive with KDAlloc. **(b)** SymLive with default allocator.

Figure 8 States observed by SymLive, using (a) KDAlloc or (b) default allocation, while executing the example from Listing 3. States are shown with heap operations (top) and allocated addresses (bottom).

Following the `then` branch keeps the number of memory objects stable; following the `else` branch leads to one leaked memory object per iteration. All paths along which `p` eventually equals a pointer returned from the `malloc` at Line 5 can form an infinite loop, where that pointer is repeatedly deallocated and allocated again at the same address. As only this address is ever deallocated, this loop requires immediate address reuse (similar programs can be crafted for any finite delay in address reuse). KDAlloc can easily detect this liveness violation when the quarantine is disabled, while KLEE's default allocator fails due to cross-path interference.

In Figure 8, we visualise the different behaviours of SymLive with KDAlloc (Figure 8a) and the default allocator (Figure 8b). In both cases, state ① corresponds to the initial `malloc` (Line 2), and shows a_1, the returned address. The symbolic state is then forked at the branch in Line 4. If the branching condition is false, the new state, state ①′, differs from state ① only by one leaked object and the constraint from the symbolic fork. On the other hand, if the condition is true, the behaviour differs between the two allocators.

In the case of KDAlloc, the allocator's internal state was forked along with the state. Thus, state ② can reuse a_1 after freeing it, even though the address is still allocated in other states. State ② therefore compares equal to state ① and the infinite loop is detected.

In contrast, the default allocator uses a single, global allocator instance, so a_1 can only be reused once it has been deallocated in every symbolic state. Since the addresses are leaked, this never happens, and state ② therefore cannot compare equal to state ①, thus no infinite loop will be detected. As a result, execution continues into state ③ (and beyond), with `malloc` returning another address in each iteration. Irrespective of the allocator, state ①′ behaves the same as state ①, as the leaked address has no further impact until all possible

allocations have been performed. Thus, with KDAlloc states (1′) and (2′) compare as equal, as do equivalent states on any further path. However, with the default allocator, states (1′) and (2′) do *not* compare as equal, nor do equivalent states on any further path.

While this example is crafted specifically to showcase the importance of cross-path determinism, it is plausible that non-deterministic allocation can cause infinite-loop bugs to be missed or at least delay their detection significantly in real-world programs. While this impact is hard to measure, given its low overhead, it is preferable to use KDAlloc with SymLive to enable or speed up infinite-loop detection.

To confirm that KDAlloc does not break any other (possibly implicit) requirement of SymLive, we ran it with KDAlloc on the applications from SymLive's artifact[13] (Table 2) where it was able to detect infinite-loop bugs. For these experiments we stayed close to the original configuration and used a memory limit of 10 GB. The experiments include `toybox`, a package that re-implements (among others) many Coreutils and `sed`. Repeating these experiments, we quickly noticed a problem with `toybox`, which failed with KDAlloc. The reason for this turned out to be simple: `toybox` uses a dispatch mechanism that allows users to call many tools through a single binary, e.g. `./toybox sed`. As part of this mechanism, `toybox` implements a safeguard that tries to detect whether the stack depth is too high. This detection, however, is implemented by comparing the integer representation of two pointers to stack variables from different stack frames, which gives an implementation-defined result as per the C standard [14]. In our design, which is fully standard-compliant in this regard, we do not follow a linear, stack-like order when allocating such variables. Additionally, we try to maximise redzones around each allocation. Together, this leads `toybox`'s safeguard to bail out when we use KDAlloc. All `toybox` utilities can also be built as standalone binaries, avoiding the dispatch along with its problematic safeguard. We used this (deviating from the original setup) for all our `toybox` runs.

Our extension was able to find infinite loops in all applications with the default exploration strategy. For the various `sed` implementations, infinite loops were only found for the shorter of two possible symbolic inputs, as, with the longer inputs, SymLive runs into memory and time limits (set to 24 h) irrespective of the allocator. We refrained from further exploring which allocator works better in that case, as KLEE frees up memory by (non-deterministically) terminating states once its memory limit is reached.

7 Related Work

As a core concept in programming, memory allocation is a well-researched topic [2,13,17,18,30]. In fact, many of the building blocks used to create KDAlloc are well-known methods in this area of research. For example, separating allocations by size to quickly find the best-fitting unallocated region [17], combining multiple equally-sized objects into one large run [13], using bitmaps to denote allocated/free slots in a larger run of equally-sized objects [3, 13], spacing objects further apart in memory [3, 25, 28], delaying deallocations in a quarantine zone [25, 28], and segregating heap data and metadata [3] are all well-established techniques for designing memory allocators.

However, to the best of our knowledge, these building blocks have never been combined in such a way, as to fit and support EGT-style dynamic symbolic execution. The closest work we see is SymMMU [24], which separates the dispatch mechanism for memory accesses in DSE from its handling policy. However, unlike KDAlloc, SymMMU is not directly concerned with determinism and stability.

[13] `https://doi.org/10.5281/zenodo.5771192`

Common memory-safety checkers, such as AddressSanitizer [28] or Valgrind Memcheck [29] have largely the same goals w.r.t. error detection via spatial and temporal distancing, but, while cross-run determinism is of some interest, they have no notion of multiple paths, which leads them to not consider cross-path determinism, stability, or a method for efficiently forking the allocator state. KDAlloc takes inspiration from these memory-safety checkers to detect faults with spatial and temporal distancing, while also considering our remaining goals, trading away performance and delegating error detection to the underlying DSE engine.

Similar questions and solutions arise when considering fault tolerance in addition to fault detection. In Rx [23], memory errors in a process are detected and as one potential mitigation strategy, allocations are moved to different locations on recovery to avoid subsequent crashes. The Windows Fault Tolerant Heap (FTH) [25] is automatically enabled when a program shows behaviour related to faults in dynamic memory management. It mitigates potential problems by utilising a quarantine and additional space between objects. Similarly, DieHard [3] randomises the heap layout over a large region to probabilistically increase both spatial and temporal distance between allocated objects.

8 Conclusion

In this paper, we show that the memory allocator can have a significant impact in dynamic symbolic execution (DSE). We first identify six key design principles – support for external calls, cross-run and cross-path determinism, spatially and temporally distanced allocations, and stability – and propose KDAlloc, a memory allocator specifically designed for DSE, whose design is guided by these principles.

We implemented KDAlloc in KLEE, a popular DSE engine, and show that it has a neutral or positive impact on memory consumption and performance, while improving use-after-free error detection and several DSE-based techniques such as MoKlee, an approach for saving DSE runs to disk and later (partially) restoring them, and SymLive, an approach for finding infinite-loop bugs.

References

1. Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proc. of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19)*, May 2019.
2. Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, November 2000.
3. Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'06)*, June 2006.
4. Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, March-April 2008.
5. Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. Memory models in symbolic execution: key ideas and new thoughts. *Software Testing, Verification and Reliability*, 29(8), 2019. doi:10.1002/stvr.1722.
6. Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)*, September 2008.

7 Frank Busse, Martin Nowack, and Cristian Cadar. Running symbolic execution forever. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)*, July 2020.

8 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

9 Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, August 2005. doi:10.1007/11537328_2.

10 Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, October 2006. doi:10.1145/1455518.1455522.

11 Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)*, 56(2):82–90, 2013. doi:10.1145/2408776.2408795.

12 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, March-April 2008.

13 Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proc. of the 2006 BSDCan Conference (BSDCan'06)*, May 2006.

14 International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.

15 Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of the 19th International Conference on Computer-Aided Verification (CAV'07)*, July 2007.

16 Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, August 2019.

17 Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

18 Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'04)*, June 2004.

19 Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

20 Martin Nowack. Fine-grain memory object representation in symbolic execution. In *Proc. of the 34th IEEE International Conference on Automated Software Engineering (ASE'19)*, November 2019.

21 Martin Nowack, Katja Tietze, and Christof Fetzer. Parallel symbolic execution: Merging in-flight requests. In *Proc. of the Haifa Verification Conference (HVC'15)*, December 2015.

22 Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)*, July 2013.

23 Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, October 2005.

24 Anthony Romano and Dawson Engler. SymMMU: Symbolically executed runtime libraries for symbolic memory access. In *Proc. of the 29th IEEE International Conference on Automated Software Engineering (ASE'14)*, September 2014.

25 Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows®Internals, Part 2.* Microsoft Press, 6th edition, September 2012.

26 Daniel Schemmel, Julian Büning, Oscar Soria Dustmann, Thomas Noll, and Klaus Wehrle. Symbolic liveness analysis of real-world software. In *Proc. of the 30th International Conference on Computer-Aided Verification (CAV'18)*, July 2018.

27 Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. Symbolic partial-order execution for testing multi-threaded programs. In *Proc. of the 32nd International Conference on Computer-Aided Verification (CAV'20)*, July 2020.

28 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, June 2012.

29 Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, April 2005.

30 Matthias Springer and Hidehiko Masuhara. DynaSOAr: A parallel memory allocator for object-oriented programming on GPUs with efficient memory access. In *Proc. of the 33rd European Conference on Object-Oriented Programming (ECOOP'19)*, July 2019.

31 David Trabish, Shachar Itzhaky, and Noam Rinetzky. Address-aware query caching for symbolic execution. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'21)*, April 2021.

32 David Trabish and Noam Rinetzky. Relocatable addressing model for symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)*, July 2020.

33 Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12)*, July 2012.

Accumulation Analysis

Martin Kellogg ✉
University of Washington, Seattle, WA, USA

Narges Shadab ✉
University of California, Riverside, CA, USA

Manu Sridharan ✉
University of California, Riverside, CA, USA

Michael D. Ernst ✉
University of Washington, Seattle, WA, USA

Abstract

A typestate specification indicates which behaviors of an object are permitted in each of the object's states. In the general case, soundly checking a typestate specification requires precise information about aliasing (i.e., an alias or pointer analysis), which is computationally expensive. This requirement has hindered the adoption of sound typestate analyses in practice.

This paper identifies *accumulation typestate specifications*, which are the subset of typestate specifications that can be soundly checked without any information about aliasing. An accumulation typestate specification can be checked instead by an accumulation analysis: a simple, fast dataflow analysis that conservatively approximates the operations that have been performed on an object.

This paper formalizes the notions of accumulation analysis and accumulation typestate specification. It proves that accumulation typestate specifications are exactly those typestate specifications that can be checked soundly without aliasing information. Further, 41% of the typestate specifications that appear in the research literature are accumulation typestate specifications.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Typestate, finite-state property

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.10

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.22`

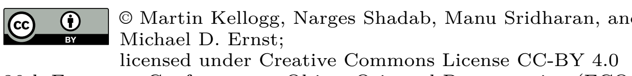
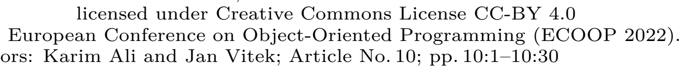
Funding This research was supported in part by the National Science Foundation under grants CCF-2007024 and CCF-2005889, DARPA contract FA8750-20-C-0226, a gift from Oracle Labs, and a Google Research Award.

Acknowledgements Thanks to Max Willsey, Gus Smith, and the anonymous reviewers for their helpful feedback on early drafts.

1 Introduction

A typestate specification [58] associates a finite-state machine (FSM) with program values of a given type. As a value transitions through the states of the FSM, different operations are enabled or disabled; that is, the FSM encodes a behavioral specification for the type.

A typestate analysis checks that a program follows a typestate specification – that is, the program does not attempt to perform a disabled operation. Typestate analyses are well-studied in the literature, and have been deployed for many purposes, including enforcing a locking discipline [28, 17], verification of Windows device drivers [12], and preventing security vulnerabilities [50]. However, *sound* typestate analyses – those with no false negatives – are rarely deployed in practice; for example, a recent paper [21] describing how AWS has deployed


© Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 10; pp. 10:1–10:30

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

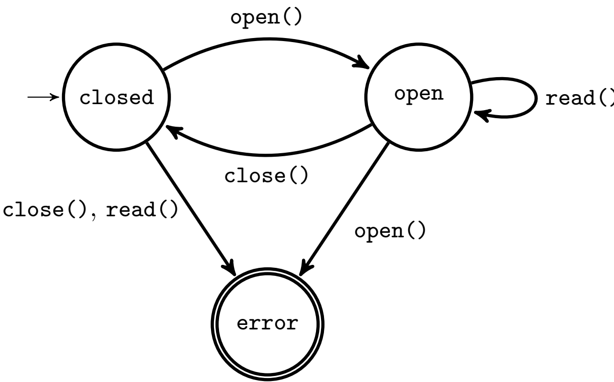


Figure 1 The typestate automaton for a `File` object that can be re-opened after being closed. This typestate specification is not an accumulation typestate system: soundly enforcing it statically requires an alias analysis.

a typestate-based analysis at cloud-scale explicitly omits soundness as a goal. However, building a sound analysis is an important goal: without a soundness guarantee, an analysis might find some bugs, but could not guarantee that no more bugs remain.

A key barrier to sound typestate analyses is the need to reason about aliasing. Consider the classic example [28, 70, 59, 25, 29, 62, 67, 57, 69, 66, 1, 49, 16, 38, 2, 15, 72, 19, 20] of a `File` object, whose typestate is specified in Figure 1, and the following program in a Java-like imperative language:

```
1    File f = new File(...);
2    f.open();
3    File g = f;  // f and g are aliases after this line is executed
4    g.close();
5    f.read();    // an error occurs when this line is executed
```

On line 3, the shared object – which both aliases `f` and `g` refer to – is in the `open` typestate. When `g.close()` is called on line 4, the state of the underlying object transitions to the `closed` state. It is therefore an error when `f.read()` is called on line 5. However, if a static typestate analysis analyzing this program does not consider that `f` and `g` are aliased, then the analysis's estimate of `f`'s typestate does not transition to the `closed` state, and the analysis unsoundly concludes that the call on line 5 is safe – that is, the analysis suffers from a false negative.

For a sound typestate analysis, there are two high-level approaches to handling aliasing: restrict how the programmer creates aliases (e.g., via ownership types [14, 55] or access permissions [7]), or use a sound inter-procedural may-alias analysis that conservatively over-approximates which program variables might be aliases. In practical imperative programming languages with unrestricted aliasing, inter-procedural may-alias analysis is NP-hard [41], and scaling alias analysis to real programs while maintaining acceptable precision remains an open research problem. State-of-the-art analyses often run for an hour or more on practical programs [60].

In recent work [35, 37], we proposed bespoke *accumulation analyses* that soundly and modularly solve specific problems traditionally addressed with typestate. An accumulation analysis collects operations – corresponding to typestate transitions – that have definitely occurred on a given program expression. For example, an accumulation analysis could check the property "before calling `read()` on a `File`, call `open()`." The accumulation analysis would

record on which expressions `open()` had definitely been called, and forbid calls to `read()` that did not occur via such expressions. Note that this is a weaker property than the full specification in Figure 1 – it does not forbid “read after close” defects.

Unlike a traditional typestate analysis, an accumulation analysis is sound without any aliasing information. This means that checking a specification with an accumulation analysis is cheaper – often by an order of magnitude or more – than checking that same specification with a general-purpose typestate analysis. Further, effective incremental analysis – i.e., modularity – is possible for an accumulation analysis, because no whole-program alias analysis is needed. Practical accumulation analyses do use limited, cheap, local aliasing information to improve precision; see Section 5.1. A practical accumulation analysis using limited aliasing information is sound because no aliasing information at all is required for soundness.

Our prior work argued informally that our accumulation analyses are sound, despite their lack of alias reasoning, due to the monotonicity of the particular typestate properties being checked. However, we neither formalized our arguments nor generalized our arguments beyond the specific problems that we targeted. Though our prior work has demonstrated good empirical results – running quickly and finding many real bugs – its soundness claim relies on accumulation analyses being sound without any aliasing information.

The primary goals of this paper are to prove that accumulation analysis does not require aliasing information, to demarcate exactly those typestate specifications that can be soundly checked via an accumulation analysis, and to explore how common such specifications are. Our hope is that analysis designers facing typestate-like problems in the future can use our work to determine whether the property they are interested in is an accumulation property, and hence could be verified without resorting to an expensive, whole-program alias analysis.

Our contributions are:

- a formal definition of an accumulation analysis (Section 3.1);
- a formal definition of an *accumulation typestate system*, and a proof that the properties checkable via accumulation analysis are all accumulation typestate properties (Section 3.2);
- a proof that a typestate system can be checked soundly by a typestate analysis that does no aliasing reasoning if and only if it is an accumulation typestate system (Section 3.3);
- a literature survey of work on typestate analysis, from which we collected 1,355 typestate specifications and determined that 41% of them are accumulation typestate specifications (Section 4); and
- a discussion of the practical issues related to implementing a useful accumulation analysis, and an implementation of a generic accumulation analysis (Section 5).

2 Background: What Is Typestate?

In a standard type system, the type of an expression is immutable throughout the program and the set of operations available on the expression is correspondingly immutable. However, type systems fail to capture the behavioral specifications of many real-world objects that change over time. For example, a chess pawn might become a queen and gain new movement operations, a caterpillar might become a chrysalis and lose the ability to crawl before eventually becoming a butterfly and gaining the ability to fly, or a `File` might be opened and gain the ability to be read. In each of these examples, the logical identity of the object stays the same, but its state – and what that state enables it to do – changes. Typestate [58] extends types to account for possible state changes by encoding the various states and behaviors of a type as a finite-state machine – the typestate automaton for that type. Formally:

▶ **Definition 1.** *A* ***typestate automaton*** $A = (\Sigma, S, s_0, \delta, e)$ *for type* τ *is a finite-state machine. The language* Σ *is the set of operations, such as method calls, that can be performed on* τ*. The states* S *are called* typestates*;* $s_0 \in S$ *is the initial state. The edges defined by the transition table* δ *are called transitions and correspond to the effect of operations. There is a distinguished error state* $e \in S$*. Each typestate has* $k = |\Sigma|$ *outgoing transitions; none, some, or all of these transitions may be to the error state* e *or may be self-loops. The error state* e *has only self-loops – that is, the error state is a trap state.*

At every step during the execution of a program, each value/object of type τ is in one of the typestates of the typestate system.

▶ **Definition 2.** *An* ***operation*** *is an event that may cause an object to change state. Every type has a set of operations that can be performed on it, but not all operations are necessarily legal in all states. Traditionally, operations are method calls. However, they can be generalized to include any other event, such as assigning a field or a reference going out of scope.*

Without loss of generality, we represent typestate automata as having no distinguished accepting states (or, equivalently, all non-error states are accepting). If a typestate automaton were to have one or more accepting states, we could transform it to have no accepting states but encode the same behavioral specification in the following way: add a "go out of scope" transition to each typestate; in accepting states (and the error state), this is a self-loop transition, but in non-accepting states, this is a transition to the error state.

▶ **Definition 3.** *A* ***typestate system*** *is the pair of a typestate automaton and the corresponding type* τ *whose safe usage it encodes.*

As an example of a typestate system, Figure 1 shows the automaton, and the type is `File`. Note how each edge is labeled with the corresponding operation. A double circle around the state represents the distinguished error state e. We always draw all transitions, with the exception of those from the error state (which are, by definition, always self-loops).

This paper considers only static typestate analyses. Dynamic run-time monitoring to detect typestate violations exists, but a run-time monitor – like any dynamic analysis – cannot prevent errors before they happen. See Section 6 for more details on related techniques that are outside the scope of the present work.

3 Definitions and Proofs

This section has three goals. First, Section 3.1 formally defines accumulation analysis in a way that is consistent with prior work. Second, Section 3.2 defines an *accumulation typestate system* and shows that every accumulation analysis has a corresponding accumulation typestate system. Finally, Section 3.3 proves that accumulation typestate systems are exactly those typestate systems that can be soundly checked by a static typestate analysis with no aliasing information – that is, a typestate-like analysis that assumes that no aliasing occurs in the program.

3.1 Accumulation Analysis

First, we formalize the notion of an accumulation analysis, as used in prior work [35, 37]:[1]

[1] Our definition is consistent with but not identical to the definitions used in prior work. See Section 6.1.

▶ **Definition 4.** *An* ***accumulation analysis*** *is a static program analysis that approximates, for each in-scope expression x of type τ at each program point, a set of operations S that have definitely occurred on the value to which x refers.*

An accumulation analysis has one or more ***goals****. A goal is a pair $\langle g, E \rangle$ where g is the* ***goal operation*** *and E is a set of* ***enabling operations****.*

Informally, an accumulation analysis enforces that a goal operation g does not occur until after every enabling operation $e \in E$ for g has already occurred.

An operation in an accumulation analysis is defined identically to an operation in a typestate automaton (Definition 2).

▶ **Definition 5.** *A* ***sound accumulation analysis*** *must issue an error if some goal operation may occur before its enabling operations. More formally, it must issue an error if, for some expression x of type τ and some operation g, both of the following are true:*

1. *There exists at least one goal $\langle g, _ \rangle$ – that is, g is a goal operation.*
2. *There exists an execution of the program where the set of operations S that have actually occurred on the value of x before an occurrence of g on x is not a superset of one of the enabling sets for g. That is, where there does not exist some goal $\langle g, E \rangle$ such that $S \supseteq E$.*

Intuitively, a sound accumulation analysis is "accumulating" enabling operations, and once everything in the enabling set is accumulated, there is no way to "disable" the goal operation. For example, if g is a goal operation for some goal $\langle g, E \rangle$, an object must first perform some set of operations to make g legal (i.e., the operations in E), and once g becomes legal, it *stays* legal.

Note that soundness, as in Definition 5, only precludes false negative warnings. It says nothing about whether the accumulation analysis might issue a false positive, and a trivially-sound "accumulation analysis" could simply issue an error any time a goal operation might be executed. In practice, a useful accumulation analysis tracks whether the transitions in an enabling set have occurred, and it permits the goal operation if they have.

Note that if an accumulation analysis has multiple goals, their goal operations may or may not be the same. Multiple goals with the same goal operation are useful to express disjunctive specifications. For example, prior work [35] used the disjunctive specification "call either `withOwners()` or `withImageIds()` before calling `describeImages()`."

3.2 Relationship Between Typestate and Accumulation

Next, we need to describe the relationship between a typestate system and an accumulation analysis. As an aid to doing so, we introduce the following:

▶ **Definition 6.** *An* ***error-inducing sequence*** *in a typestate automaton T is a sequence of transitions $S = t_1, \ldots, t_i$ such that T is in the error state after all transitions in S are applied (and not before).*

▶ **Definition 7.** *An* ***accumulation typestate system*** *is a typestate system such that for any error-inducing sequence $S = t_1, \ldots, t_i$, all subsequences (including both contiguous and non-contiguous subsequences) of S that end in t_i also result in the typestate automaton being in the error typestate. That is, all subsequences of S that end in t_i are also error-inducing.*

Intuitively, an accumulation typestate system is any typestate system whose error-inducing paths are closed under subsequence so long as the final error-inducing operation is held constant. That is, removing operations from the beginning or middle of an error-inducing sequence always produces another error-inducing sequence.

Algorithm 1 A decision procedure for checking whether or not a given typestate automaton T is an accumulation typestate automaton. The complexity of the algorithm is $O(max(n \log n, en))$ where n is the number of states and e is the number of edges.

```
1:  procedure IsAccumulation(T)
2:      // FindErrorInducingTransitions returns all transitions into the error state.
3:      U ← FindErrorInducingTransitions(T)
4:      // E and E_subseq are finite-state automata. ∀ X, Union(∅, X) = X.
5:      E ← ∅
6:      E_subseq ← ∅
7:      for u_i ∈ U do
8:          // ErrorInducingAutomatonVia is an automaton that accepts a sequence of
9:          // transitions S iff S followed by u_i causes an error in the original automaton T.
10:         // Its implementation contains two steps: (1) modify T so that states from which
11:         // u_i is error-inducing are accepting, and then (2) minimize and return the result.
12:         E_i ← ErrorInducingAutomatonVia(u_i, T)
13:         // Subsequences produces the automaton that accepts the subsequence language
14:         // for the input automaton, which Higman's theorem guarantees exists.
15:         E_subseq(i) ← Subsequences(E_i)
16:         // Concat produces an automaton that accepts iff it receives a sequence
17:         // that the input automaton accepts followed by the concatenated transition.
18:         E ← Union(E, Concat(E_i, u_i))
19:         E_subseq ← Union(E_subseq, Concat(E_subseq(i), u_i))
20:     // AcceptSameLanguage is true iff the two automata accept the same language.
21:     return AcceptSameLanguage(E, E_subseq)
```

Note that a vacuous sound typestate analysis such as "issue an error at every program statement" is trivially enforcing an accumulation typestate system. The typestate automaton that such an analysis enforces only has transitions to the error state, so all sequences are error-inducing.

This definition leads to a decision procedure (Algorithm 1) for determining whether a given typestate system T is an accumulation typestate system. Consider all error-inducing operations $U = \{u_1, \ldots, u_n\}$. The elements of U are the final transitions for every error-inducing sequence in the automaton of T. For any $u_i \in U$, let E_i be the language[2] of the error-inducing sequences of operations in T that end in u_i, with the last transition removed (i.e., the u_i transition that leads to the `error` typestate). Let $E_{subseq(i)}$ be the language of subsequences of E_i. Let $E = \bigcup_{i=1}^{n} E_i * u_i$ and $E_{subseq} = \bigcup_{i=1}^{n} E_{subseq(i)} * u_i$. That is, E is the union of all error-inducing paths in T, and E_{subseq} is the union of all subsequences of error-inducing paths in T that end in the same transition as the corresponding error-inducing path from which they were derived. By Definition 7, if and only if E and E_{subseq} recognize the same language, T is an accumulation typestate system.

It is easy to check whether E and E_{subseq} recognize the same language, because both are regular. E is regular, because it can be recognized by T's automaton, if the error typestate is converted to an accepting state. Since there are finitely-many operations, any E_i and $E_{subseq(i)}$ have a finite alphabet. Higman's theorem [31] says that the language of the subsequences

[2] Throughout, we will abuse notation and refer to both languages and their corresponding language-recognizers by the same name.

of any language over a finite-alphabet is regular. Therefore, any $E_{subseq(i)}$ is also regular. E_{subseq} is regular because regular languages are closed under both union and concatenation. So, the procedure for checking whether a typestate automaton is an accumulation typestate automaton is as easy as checking whether the two finite state machines for E and E_{subseq} recognize the same language.

▶ **Theorem 8.** *Every accumulation analysis has a corresponding accumulation typestate system.*

Proof. Consider some accumulation analysis *acc* with goals $(g_1, E_1), \ldots, (g_n, E_n)$ over type τ. The corresponding accumulation typestate system is the pair of the type τ and the accumulation typestate automaton constructed by the following procedure:

1. Create an error state `error` with a self-loop transition for each operation on τ.
2. Let $\mathcal{P}_E$ be the powerset of E, where $E = \bigcup_{i=1}^{n} E_i$ is the union of the enabling sets $E_1, \ldots, E_n$. For each element S of $\mathcal{P}_E$, create a corresponding state and label it with S. Note that S refers to both the member of $\mathcal{P}_E$ and the corresponding state.
3. Make the state that is labeled by the empty set be the start state of the automaton.
4. For each state $S \in \mathcal{P}_E$ and for each transition $t_e \in E$, add a transition from state S to state $S \cup \{t_e\}$ labeled t_e. (This transition might be a self-loop.)
5. Let $G = \{g_1, \ldots, g_n\}$ be the set of goal transitions. For each element g_i of G and for each state $S \in \mathcal{P}_E$:
   - If there exists a goal $\langle g_i, E_i \rangle$ such that $E_i \subseteq S$,
     - then add a self-loop transition to S labeled g_i if it does not already have a transition labeled g_i. (It might have such a transition if g_i is both an enabling transition and a goal transition.)
   - Else if such a goal does not exist,
     - add a transition from S to the `error` state labeled g_i, removing a transition labeled g_i if one already exists.
6. For each operation t on τ such that $t \notin G$ and $t \notin E$ – that is, for each operation that is neither a goal operation nor an enabling operation – add a self-loop transition labeled t to each non-error state. (Recall that the error state already has self-loop transitions for each operation, added in step 1.)

The resulting accumulation typestate automaton encodes the same behavior as the original accumulation analysis. ◀

Note that this construction is a existence proof, not an efficient translation: it does induce an exponential blowup in the number of states. A practical accumulation analysis does not track states directly – rather, it tracks only the enabling sets – so state explosion is not a problem in practice.

3.3 Soundness Without Aliasing

This section proves that accumulation typestate systems are exactly the typestate systems that are soundly checkable without reasoning about aliasing (i.e., by a *typestate analysis with no aliasing information*, which we will formally define in Definition 14):

▶ **Theorem 9.** *A typestate system $T = (A, \tau)$ is an accumulation typestate system if and only if there exists a typestate analysis with no aliasing information that can soundly check T.*

The high-level intuition behind the proof of Theorem 9 is the consequence of two facts:

- without using aliasing information, a typestate analysis observes only a subsequence of the actual operations that are applied to the object to which some expression refers, and
- accumulation typestate automata are exactly those that are error-closed under subsequence, when the last transition is held constant.

The formal proof is split into Lemmas 16 and 17 (which are the forward and backward directions of the bi-implication respectively), and appears in Section 3.3.2. Section 3.3.1 defines the supporting machinery of the proof: the language, relevant definitions, etc.

Accumulation analyses as defined in Section 3.1 (and therefore as defined in prior work [35, 37]) are sound without access to aliasing information:

▶ **Corollary 10.** *An accumulation analysis, even without aliasing information, is sound.*

Proof. Convert the accumulation analysis to an accumulation typestate system via the procedure in the proof of Theorem 8. By Theorem 9, the accumulation typestate system can be soundly checked. ◀

An important consequence of the ability to soundly check an accumulation typestate system with *no* aliasing information is that approaches that utilize *limited* aliasing information are also sound. In practice, analyses can compute inexpensive, typically local, alias information to improve precision (i.e., to avoid issuing false positive warnings); see Section 5.1.

3.3.1 Preliminaries

This section introduces the machinery used to prove Theorem 9.

3.3.1.1 Language

We will prove Theorem 9 over a core calculus that represents a simple imperative programming language. This language contains the essential parts of a programming language related to typestate checking and aliasing – method calls, fields, and assignments.

A program P in this language is a statement s of one of the following kinds:

- an assignment: x_i `:=` x_j.
- a field load: x_i `:=` x_j`.`f_k.
- a field store: x_i`.`f_j `:=` x_k.
- a method call: x_i`.`m_j`()`.
- a statement sequence: s_i `;` s_j.

Source code variables range from $\texttt{x_1}$ to $\texttt{x_}n$, where n is some positive integer. Statements may only refer to variables in that range. There is a single type T. Each variable refers to a *value* – that is, a particular object instance – of type T. We use $x_i, x_j, \ldots$ as metavariables for arbitrary variables in the range $\texttt{x_1},\ldots,\texttt{x_}n$. T has methods $\texttt{m_1}$ to $\texttt{m_}k$ and a corresponding typestate automaton A whose k operations are exactly the methods $\texttt{m_1}$ to $\texttt{m_}k$. A method call statement can only refer to methods in T. We use $m_i, m_j, \ldots$ as metavariables for arbitrary methods in T. Each object of type T has fields $\texttt{f_1}$ to $\texttt{f_}m$, where m is some positive integer. Load and store statements may only refer to fields in this range. Each field refers to some value of type T. We use $f_i, f_j, \ldots$ as metavariables for arbitrary fields in T.

To simplify the presentation and proofs, this language lacks conditionals, loops, method bodies, return values, etc. – which makes precise alias and typestate analysis trivial. However, our algorithms are general (they do not take advantage of the straight-line nature of the code) and can be extended to a richer language without changing the essence of the proof. Section 5.2 discusses practical concerns when implementing an accumulation analysis for a real programming language.

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j \Downarrow \langle \rho[x_i \mapsto \rho(x_j)], \sigma, \tau \rangle} \text{ ASSIGN}$$

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j.f_k \Downarrow \langle \rho[x_i \mapsto \sigma(\langle \rho(x_j), f_k \rangle)], \sigma, \tau \rangle} \text{ LOAD}$$

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i.f_j := x_k \Downarrow \langle \rho, \sigma[\langle \rho(x_i), f_j \rangle \mapsto \rho(x_k)], \tau \rangle} \text{ STORE}$$

$$\frac{\langle \rho, \sigma, \tau \rangle \vdash t' = \mathit{succ}(\tau(\rho(x_i)), m_j, A) \qquad t' \neq \texttt{error}}{\langle \rho, \sigma, \tau \rangle \vdash x_i.m_j() \Downarrow \langle \rho, \sigma, \tau[\rho(x_i) \mapsto t'] \rangle} \text{ CALL}$$

$$\frac{\langle \rho, \sigma, \tau \rangle \vdash s_i \Downarrow \langle \rho', \sigma', \tau' \rangle \qquad \langle \rho', \sigma', \tau' \rangle \vdash s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle}{\langle \rho, \sigma, \tau \rangle \vdash s_i; s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle} \text{ SEQ}$$

Figure 2 The big-step dynamic semantics of the language expressed as inference rules. The notation $\mu[x \mapsto y]$ means that the map μ is updated so that x maps to y. $M \vdash s \Downarrow M'$ means that executing statement s in machine-state M results in machine-state M'.

3.3.1.2 Dynamic Semantics

To execute a program, we maintain a *machine state* $\langle \rho, \sigma, \tau \rangle$ composed of an environment (ρ) mapping each variable to a value of type T, a store (σ) mapping each value–field pair to a value, and a typestate store (τ) mapping each value to a typestate in A. The initial environment maps each x_i to a distinct value v_j. The initial store maps each value–field pair $\langle v_i, f_j \rangle$ to a distinct value v_k. The initial typestate store maps each value v_i to the start typestate s_0 of A.[3] Executing a statement in machine state $\langle \rho, \sigma, \tau \rangle$ either produces an updated machine state $\langle \rho', \sigma', \tau' \rangle$, or it terminates the program in an error if any value's entry in the typestate store would be A's `error` typestate. The dynamic semantics (Figure 2) are as follows:

- For an assignment x_i `:=` x_j, produce a new machine state with an updated environment: $\rho'(x_i) = \rho(x_j)$ (rule ASSIGN).
- For a field load x_i `:=` x_j`.`f_k, produce a new machine state with an updated environment: $\rho'(x_i) = \sigma(\rho(x_j), f_k)$ (rule LOAD).
- For a field store x_i`.`f_j `:=` x_k, produce a new machine state with an updated store: $\sigma'(\rho(x_i), f_j) = \rho(x_k)$ (rule STORE).
- For a call x_i`.`m_j`()`, let $t' = \mathit{succ}(\tau(\rho(x_i)), m_j, A)$. That is, t' is the successor typestate in A when transition m_j occurs in the current typestate of the value that x_i is a reference to. If t' is not the `error` typestate, produce a new machine state with an updated typestate store: $\tau'(\rho'(x_i)) = t'$ (rule CALL). If t' is the `error` typestate, the semantics "get stuck" and the program terminates in an error.
- For a sequence s_i `;` s_j, first execute s_i. If the program terminates in an error while executing s_i, the semantics for the sequence statement "get stuck." Otherwise, let $\langle \rho', \sigma', \tau' \rangle$ be the machine state after executing s_i. Execute s_j in $\langle \rho', \sigma', \tau' \rangle$ (rule SEQ).

[3] Initializing all variables before a program starts simplifies the language by removing the need for a `new` expression.

3.3.1.3 Sound Typestate Analysis

▶ **Definition 11.** *A* ***typestate analysis*** *is a static program analysis. Its inputs are a program P and a typestate system $T = (A, \tau)$. It reports call statements within P that may cause the program to terminate in an error when running P.*

▶ **Definition 12.** *A typestate analysis is* ***sound*** *if it reports each call statement that causes the program to terminate in an error at run time in any execution of the program.*

3.3.1.4 Representation of Aliasing

Suppose that a typestate analysis has access to two oracle functions $\mathit{MustOracle}(x_i, s)$ and $\mathit{MayOracle}(x_i, s)$ for aliasing information. Each oracle takes a variable x_i and a program statement s and returns a list of *names* – variables or arbitrarily-nested field load expressions – that the input variable must (respectively, may) alias before the given statement.

MustOracle returns a list of names that definitely do alias x_i at s. More formally, for a sound oracle, if the list returned by $\mathit{MustOracle}(x_i, s)$ contains x_j, then x_i and x_j are definitely aliased before statement s on all executions. If the list does not contain x_j, then x_i and x_j may or may not be aliased before s. A trivial *MustOracle* that always returns an empty list is sound.

MayOracle returns a list of names that *might or might not* alias x_i at s. More formally, for a sound oracle, if the list returned by $\mathit{MayOracle}(x_i, s)$ does not contain x_j, then x_i and x_j are definitely not aliased before statement s on all executions. If the list does contain x_j, then x_i and x_j may or may not be aliased before s. A trivial *MayOracle* that always returns every in-scope name in the program is sound.

These oracles can represent an external alias analysis, an on-demand alias analysis, aliasing tracking built into the typestate analysis, etc. If the oracles are sound, then for all x_i and s, $\mathit{MustOracle}(x_i, s) \subseteq \mathit{MayOracle}(x_i, s)$. For a traditional typestate analysis (as defined in section 3.3.1.5) to be sound for an arbitrary typestate system such as the `File` example in Figure 1, both oracles must be sound.[4]

3.3.1.5 Definition of Typestate Analysis

A typestate analysis is a fixpoint analysis that can be viewed as a dataflow analysis or an abstract interpretation. It operates by maintaining a set of *abstract stores*, one for each program point. An abstract store is a map from names to sets of estimated typestates. We write $\phi_s(x_i)$ for the estimated typestates of name x_i before program statement s, and $\phi'_s(x_i)$ for those after. For any sequencing statement $r;s$, for all x_i, $\phi'_r(x_i) = \phi_s(x_i)$. The notation $\hat{\phi}_s(x_i.*)$ means all names in ϕ_s that begin with x_i.

At the beginning of the analysis, at every program point, the abstract store maps all names[5] to the set containing only the start state s_0 of the typestate automaton A. Then, the analysis processes each statement s using the following rules (which also appear in Figure 3) until the set of abstract stores reaches a fixpoint:

---

[4] For the language of section 3.3.1.1, it is trivial to construct a sound alias analysis that never includes a name in the result of a *MayOracle* query unless the corresponding *MustOracle* query would also include that name. In a richer programming language, the *MayOracle* is necessary to handle analysis imprecision and control flow joins.

[5] An analysis may use widening, abstraction, or iterative expansion of maps to handle the fact that the set of names is infinite.

$$\frac{\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j/x_i] \wedge T'_{n'} = \phi_s(n') \quad \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]}{\phi_s \vdash x_i := x_j \Downarrow \phi'_s} \text{ TS-ASSIGN}$$

$$\frac{\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j.f_k/x_i] \wedge T'_{n'} = \phi_s(n') \quad \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]}{\phi_s \vdash x_i := x_j.f_k \Downarrow \phi'_s} \text{ TS-LOAD}$$

$$\frac{\begin{array}{c}\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.f_j.*), n' = n[x_k/x_i.f_j] \wedge T'_{n'} = \phi_s(n') \wedge \\ A_n^{must} = \mathit{MustOracle}(n, s) \wedge A_n^{may} = \mathit{MayOracle}(n, s) \\ \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.f_j.*), n \mapsto T'_{n'}][\forall a_n \in A_n^{must}, a_n \mapsto T'_{n'}] \\ [\forall b_n \in A_n^{may} - A_n^{must}, b_n \mapsto T'_{n'} \cup \phi_s(b_n)]\end{array}}{\phi_s \vdash x_i.f_j := x_k \Downarrow \phi'_s} \text{ TS-STORE}$$

$$\frac{\begin{array}{c}\phi_s \vdash T = \phi_s(x_i) \qquad T' = \bigcup_{t \in T} \mathit{succ}(t, m_j, A) \\ A^{must} = \mathit{MustOracle}(x_i, s) \qquad A^{may} = \mathit{MayOracle}(x_i, s) \\ \phi'_s = \phi_s[x_i \mapsto T'][\forall a \in A^{must}, a \mapsto T'][\forall b \in A^{may} - A^{must}, b \mapsto T' \cup \phi_s(b)]\end{array}}{\phi_s \vdash x_i.m_j() \Downarrow \phi'_s} \text{ TS-CALL}$$

$$\frac{\phi_s \vdash s_i \Downarrow \phi'_{s_i} \qquad \phi'_{s_i} = \phi_{s_j} \qquad \phi_{s_j} \vdash s_j \Downarrow \phi'_s}{\phi_s \vdash s_i; s_j \Downarrow \phi'_s} \text{ TS-SEQ}$$

Figure 3 Inference rules for a traditional, sound typestate analysis. Each rule applies to some statement s, which appears in the consequent. The notation $x[y/z]$ means "x with each z replaced by y." The notation $\hat{\phi}_s(x_i.*)$ means all names in ϕ_s that begin with x_i.

- For an assignment x_i := x_j, for each $n \in \hat{\phi}_s(x_i.*)$, let $n' = n[x_j/x_i]$ – that is, n' is n with its x_i replaced by x_j – and let $T'_{n'} = \phi_s(n')$, the abstract value of n' in the pre-state. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$ (rule TS-ASSIGN). For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.*)$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.
- For a load statement x_i := $x_j.f_k$, for each $n \in \hat{\phi}_s(x_i.*)$, let $n' = n[x_j.f_k/x_i]$ and let $T'_{n'} = \phi_s(n')$. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$ (rule TS-LOAD). For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.*)$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.
- For a store statement $x_i.f_j$:= x_k, for each $n \in \hat{\phi}_s(x_i.f_j.*)$, let $n' = n[x_k/x_i.f_j]$ and let $T'_{n'} = \phi_s(n')$. Then, for each n and its n' and $T'_{n'}$, the analysis performs the following steps (rule TS-STORE):
  1. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$.
  2. The analysis queries $\mathit{MustOracle}(n, s)$ (call the result A_n^{must}). For each $a_n \in A_n^{must}$, the analysis performs a *strong update* to the abstract store: $\phi'_s(a_n) := T'_{n'}$.
  3. The analysis queries $\mathit{MayOracle}(n, s)$ (call the result A_n^{may}). For each element b_n in $A_n^{may} - A_n^{must}$ – that is, variables that may be aliases but are not guaranteed to be aliases – the analysis performs a *weak update* to the abstract store so that it maps b_n to $T'_{n'} \cup \phi_s(b_n)$: $\forall b_n \in A_n^{may} - A_n^{must}, \phi'_s(b_n) := T'_{n'} \cup \phi_s(b_n)$.

For all other names m in ϕ_s where $m \notin \hat{\phi_s}(x_i\,.\,f_j.*) \wedge \forall A_n^{may}, m \notin A_n^{may}$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.

- For a call statement $x_i\,.\,m_j\,()$, let $T' = \bigcup_{t \in \phi_s(x_i)}$. The analysis performs the following steps (rule TS-CALL):
  1. If any $t' \in T'$ is `error`, the analysis reports an error for the statement. Note that while the dynamic semantics (Figure 2) do not permit any value to be in the `error` typestate (the program crashes instead), this analysis approximates the semantics statically.
  2. The analysis updates the abstract store so that $\phi'_s(x_i) := T'$.
  3. The analysis queries *MustOracle*(x_i, s) (call the result A^{must}). For each $a \in A^{must}$, the analysis performs a strong update to the abstract store: $\phi'_s(a) := T'$.
  4. The analysis queries *MayOracle*(x_i, s) (call the result A^{may}). For each $b \in A^{may} - A^{must}$, the analysis performs a weak update to the abstract store: $\phi'_s(b) := T' \cup \phi_s(b)$.
- For a sequence $s = s_i\ ;\ s_j$, the analysis first analyzes s_i, and then analyzes s_j with the resulting abstract store (rule TS-SEQ)). (Note that the analysis does not terminate in the case of an error, but keeps reporting errors on subsequent statements.)

This standard formulation of a traditional typestate analysis is sound for any arbitrary typestate system, as long as its aliasing oracles are sound:

▶ **Theorem 13.** *A traditional typestate analysis is sound if its MustOracle and MayOracle functions return sound results.*

Proof. By co-induction on the dynamic semantics (Figure 2) and the rules for a traditional typestate analysis (Figure 3). The key invariant is that the actual typestate to which a name refers on any particular execution at some statement is always in the abstract store. ◀

3.3.1.6 Typestate Analysis with No Aliasing Information

▶ **Definition 14.** *A* ***typestate analysis with no alias information*** *is a typestate analysis whose MustOracle and MayOracle functions return empty lists for all arguments.*

Intuitively, a typestate analysis "with no alias information" assumes that no aliasing occurs in the program – even when making such an assumption is unsound.

A typestate analysis with no alias information has a simpler method call rule: it never updates its abstract store in response to an aliasing query, so steps 3 and 4 may be omitted. Similarly, there is a simpler store rule: only the $n \in \hat{\phi_s}(x_i\,.\,f_j.*)$ need to be updated, because all *MayOracle* and *MustOracle* queries (unsoundly) return false.

Informally, having no aliasing information means that the analysis might not be aware that one or more transitions have occurred on the value to which some expression refers, because those operations occurred via an alias. That is, the analysis's estimate of the typestate of an expression that actually refers (at run time) to a value v in typestate t is must include a typestate reachable by a subsequence of the sequence of transitions that results in $\tau(v)$ being t. Stated more formally:

▶ **Lemma 15.** *Let $R = \phi_s(x_i)$ be the set of estimated typestates produced by a typestate analysis with no aliasing information for a variable x_i before a statement s. Let S be the trace of an arbitrary execution leading up to some occurrence of s, and let $t = \tau(\rho(x_i))$ be the typestate of the actual value to which x_i refers before that occurrence of s. Applying S to the automaton leads to typestate t. There exists a typestate $r \in R$ such that applying some subsequence of S leads to r. That is, there is some estimated typestate $r \in R$ that is reachable by a subsequence of the transitions that lead to t.*

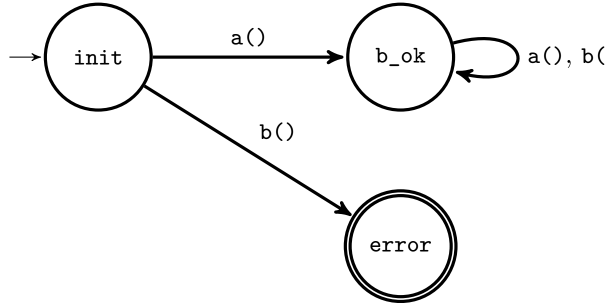


Figure 4 An accumulation typestate automaton for the property "call a() before calling b()".

$$\frac{\phi_s \vdash \phi'_s = \phi_s[x_i \mapsto s_0]}{\phi_s \vdash x_i := x_j.f_k \Downarrow \phi'_s} \text{ TS-LOAD-FIX}$$

Figure 5 A modified load rule for a typestate analysis with no aliasing information, which preserves Lemma 15. s_0 is the start state of the automaton A being checked.

Stated another way, Lemma 15 says that for every possible trace S through the program that reaches s, there is at least one $r \in R$ that "corresponds to" S, in the sense that r is reachable by a subsequence of S.

Lemma 15 is not quite true of a typestate analysis as defined in Figure 3: field loads do not necessarily preserve it. Because the store rule is unsound due to the unsoundness of the aliasing oracles, the entry in the abstract store for a given field may not actually be related to the value to which that name refers, due to possible aliasing. For example, consider the following program, being analyzed with respect to the "only call b() after a()" typestate automaton in Figure 4 (note that "Estimated state" and "Actual state" columns only show entries for names that are relevant to the problem):

Program	Estimated state (ϕ_s)[6]	Actual state (τ)[7]
`x2 = x1`	{`x1.f`↦`init`, `x2.f`↦`init`}	{`x1.f`↦`init`, `x2.f`↦`init`}
`x3.a()`	{`x1.f`↦`init`, `x2.f`↦`init`, `x3`↦`b_ok`}	{`x1.f`↦`init`, `x2.f`↦`init`, `x3`↦`b_ok`}
`x1.f = x3`	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x3`↦`b_ok`}	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x3`↦`b_ok`}
`x2.f = x4`	{`x1.f`↦`b_ok`, `x2.f`↦`init`}	{`x1.f`↦`init`, `x2.f`↦`init`}
`x5 = x1.f`	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x5`↦`b_ok`}	{`x1.f`↦`init`, `x2.f`↦`init`, `x5`↦`init` }
`x5.b()`	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x5`↦`b_ok`}	{`x1.f`↦`init`, `x2.f`↦`init`, `x5`↦`init` }

This program (left side of the table above) leads to Lemma 15 being untrue at the final statement, because the actual state of `x5` (`init`) is not reachable from the estimated state (`b_ok`). The key issue is aliasing: `x1` and `x2` are aliases, so `x1.f` and `x2.f` actually refer to the same value. When `x2.f` is re-assigned to `x4`, the actual value to which `x1.f` refers changes – but with no aliasing information, the typestate analysis is unaware, leading to the problem.

Note that this problem applies to arbitrary typestate systems: both accumulation typestate systems and non-accumulation typestate systems. Lemma 15 discusses both.

[6] Entries in ϕ_s are single-element sets. For simplicity of presentation, set notation has been elided.
[7] Keys in τ are values. For simplicity of presentation, the necessary lookups in ρ and σ have been elided.

There is a simple solution to this problem that makes Lemma 15 hold for a typestate analysis with no aliasing information: update the load rule so that the analysis assumes that all loads return a value whose typestate is the start state of the automaton (rule TS-LOAD-FIX in Figure 5).

This rule trivially preserves Lemma 15 for field loads, and corresponds with how accumulation analyses handle field loads in practice (see Section 5.2). Our proof assumes this simpler load rule for the typestate analysis with no aliasing information. However, note that this rule would make a traditional typestate analysis unsound (i.e., this rule makes Theorem 13 untrue): in an arbitrary typestate analysis, the start state is not necessarily a safe default assumption. A useful property of accumulation typestate automata, however, is that every operation which might ever lead to an error on any path must necessarily lead to an error from the start state – otherwise, the definition of accumulation typestate automaton could not be met when considering the empty subsequence.

We now prove Lemma 15 (see Appendix A for the full proof):

Proof. By co-induction on the dynamic semantics and the rules for a typestate analysis with no aliasing information. The interesting cases are method calls, assignments, and loads. Method calls preserve the inductive invariant via the inductive hypothesis. Assignments preserve the inductive invariant because the left-hand side's estimate is updated to the right-hand side's estimate, which also preserves the invariant by the inductive hypothesis. Loads preserve the inductive invariant only because of the modified rule described above, which says that after a load, the estimate is always the start state, which trivially preserves the invariant. ◀

3.3.2 Proof of Theorem 9

The proof is split into two parts – the forwards and backwards direction of the bi-implication, which are Lemmas 16 and 17, respectively.

▶ **Lemma 16.** *T is an accumulation typestate system $\implies$ there exists a sound typestate analysis with no aliasing information that can check T.*

Proof. The proof is by contradiction. Suppose that an arbitrary typestate analysis with no aliasing information (as defined by Definition 14) for an accumulation typestate system T is unsound. That is, suppose that it fails to issue an error at some method call statement $s = x_i.m_j()$, but the program terminates in an error in some execution e, because $\tau(\rho(x_i))$ after s would be `error`.

Let $v_i = \rho(x_i)$. That is, x_i actually refers to v_i at[8] s on execution e. m_j must be the transition that would lead v_i to enter the error typestate at the call $x_i.m_j()$, because the program would have already terminated if some other transition might have caused v_i to enter the error state before s was reached. Let $R' = \phi'_s(x_i)$ be the analysis's estimate of the possible typestates of x_i after the call statement is executed. Because the analysis did not issue an error at s, R' must not contain the `error` typestate.

Since R' does not contain the error typestate after observing m_j, then m_j must have been a legal transition on each typestate in the analysis' pre-state estimate $R = \phi_s(x_i)$. By Lemma 15, there is some typestate $r \in R$ that is reachable via some subsequence of the transitions that led to the actual typestate $t = \tau(\rho(x_i))$ that v_i was in during e before transition m_j was applied.

[8] s must be a method call statement, so v_i is the same before and after s.

The typestate r is reachable by a subsequence of the sequence of transitions that actually occurred on v_i that led it to reach t, but m_j is a legal transition in r. This is a contradiction: m_j must be both an error-inducing and a legal transition in r. m_j must be an error-inducing transition in r by the definition of an accumulation typestate system (Definition 7): m_j must be an error-inducing transition in typestates reachable via subsequences of the transitions that lead to t, including r. But, m_j must also be a legal transition in r because the analysis did not issue an error when its estimate included r. Since one transition cannot be both error-inducing and legal, by contradiction, the analysis must have been sound. ◀

▶ **Lemma 17.** *T is an accumulation typestate system $\Longleftarrow$ there exists a sound typestate analysis with no aliasing information that can check T.*

Proof. The proof is by contradiction. Suppose that there is a typestate analysis with no aliasing information that can soundly check a typestate system T that is not an accumulation typestate system. Since T is not an accumulation typestate system, there exists some sequence of transitions $S = t_1, \ldots, t_i$ that ends in an error typestate that has a subsequence S' that ends in t_i that does not end in an error typestate. Let D be the difference between S' and S: the sequence of transitions that appear in S but do not appear in S'.

Construct a program P with two variables $x_{S'}$ and x_D. The first statement in P is x_D `:=` $x_{S'}$, which aliases these expressions. Then augment the program in the following manner: for each transition $t \in S$, if t is an element of S', then add the statement $x_{S'}$`.`t`()` to P. Otherwise, add the statement x_D`.`t`()` to P.

Because $x_{S'}$ and x_D were aliased by P's first statement, we know that they both point to a single value v to which every transition in S has been applied by the end of P; thus, P terminates in an error when the final transition t_i is applied. However, no error is issued: the analysis will not issue an error for $x_{S'}$`.`t_i`()`, which is the program statement that causes the error, because the sequence R that was applied to $x_{S'}$ is a legal sequence of transitions (and the error-inducing transition t_i is guaranteed to be in S', not in D, by definition). This is a contradiction of our original premise that a typestate analysis with no aliasing information could soundly check T: an error-inducing transition (t_i) occurs, but the analysis with no aliasing information fails to issue an error. Thus, T must have been an accumulation typestate system. ◀

3.4 Discussion: Accumulating Sets vs. Accumulating Subsequences

Section 3 uses the term "accumulation" to refer to two subtly different things. Accumulation analyses (Definition 4) compute *sets* of operations. Accumulation typestate systems (Definition 7) are defined by *(sub)sequences* of operations.

Definition 4 of accumulation analysis uses sets because that is how accumulation analysis is defined and implemented in prior work [35, 37]. For an alternate definition of accumulation analysis in terms of subsequences, each goal operation would have an enabling sequence rather than an enabling set. Implementing an accumulation analysis based on this alternate definition would allow us to check "accumulation-like" properties that cannot be expressed as sets. For example, such an analysis could soundly check a property such as "call `a()` at least twice before calling `b()`" (i.e., a goal transition enabled by counting) or a property such as "call `a()` and `b()`, in that order, before calling `c()`" (i.e., a goal transition enabled by ordering). This generalization of the *concept* of accumulation from the specific accumulation analyses used in prior work is one of our contributions.

In our literature survey (Section 4), we found three specifications with a goal transition enabled by ordering, but we did not find any enabled by counting. For example, in Figure 12 of [56], the authors describe a mined typestate specification for the Java KeyAgreement

API. This API contains a method `generateSecret()`. Calling `generateSecret()` before `init()` and `doPhase()` is an error, so `generateSecret()` is a goal transition. However, `init()` and `doPhase()` also must be ordered: calling `doPhase()` before `init()` is also an error. The other two specifications in the literature (which appear in [56, 22]) that rely on ordering had a similar character to this example: describing some multi-stage initialization property where the initialization steps must be performed in some specific order.

4 Literature Survey

This section aims to answer the research question: **RQ1: What fraction of typestate problems can be solved modularly with an accumulation analysis?**

We will approximate the answer by using the population of typestate problems that appear in the scientific literature. Note that this is likely to be an under-approximation of incidence in practice, because scientific papers usually address the most complex problems.

We performed a literature survey of papers in the research literature since 2000 that contain typestate specifications. We chose the year 2000 because a similar survey [18], which we discuss in section 4.2.2.1, was published in 1999. For each typestate specification that we discovered, we used the decision procedure in Algorithm 1 to determine whether the specification was an accumulation typestate system – and therefore soundly analyzable without any aliasing information by Theorem 9. The vast majority of the papers that we analyzed use typestate for some small number of examples. We report on these papers in aggregate and describe specific, common examples (Section 4.2.1). There are two outliers [18, 4] that reported on categories containing hundreds of specifications, which we discuss in detail (Section 4.2.2).

The remainder of this section details our methodology, discusses the results, and gives examples of specifications that can and cannot be checked via accumulation.

4.1 Methodology

We searched Google Scholar for papers since 2000 whose full-text includes "typestate", resulting in 1,760 hits. (We originally included "type-state" and "type state" as search terms, but discovered no computer science results in the first 100 hits for each that "typestate" did not also return.) We discarded any paper that was not published in the research track of a reputable computer science conference or journal or was duplicative with another paper in the dataset (e.g., for work with both a conference paper and a journal extension, we only included the journal extension), resulting in a set of 187 papers. The authors are familiar with the relevant conferences and journals in programming languages and software engineering, and we used our judgment for these, erring on the side of inclusivity. For conferences or journals outside PL and SE, we included papers in any venue with a CORE ranking of A or A*.

We then examined each of the remaining papers in detail and recorded how many typestate specifications they contained, which specifications those were, and which of the specifications were accumulation typestate systems. When recording which specifications occurred in each paper we examined, we also recorded whether the specifications were duplicates of specifications that appeared in other papers. Among the papers we examined, 102 ($\approx 55\%$ of those examined closely, and $\approx 6\%$ of all Google Scholar hits) contained one or more typestate specifications. The venues that contributed papers with one or more typestate specifications to this study are: ECOOP (12), ESEC/FSE (12), ICSE (12), OOPSLA (10), PLDI (8), ISSTA (7), ASE (6), POPL (5), CCS (4), SAS (4), TOSEM (4), TSE (4), CC (2), ASPLOS (1), CAV (1), EuroSys (1), ICPC (1), IWACO (1), SAC (1), SOSP (1), TOPLAS (1), VMCAI (1), WWW (1).

Table 1 The results of the literature survey. "TSA" stands for "TypeState Automata"; "ATSA" stands for "Accumulation TypeState Automata". All specification counts are without de-duplication.

Dataset	Source	TSA	ATSA	ATSA%
Papers since 2000 with <20 TSAs	101 scientific papers	302	67	22%
Dwyer et al. (1999) [18]	34 papers, tools, students	511	306	60%
Beckman et al. (2011) [4]	4 real Java projects	542	182	34%
Total	All of the above	1355	555	41%

4.2 Results

Table 1 summarizes the results. This paper's artifact[9] contains our analysis of each relevant paper. The artifact also contains a finite-state machine for each typestate problem (as defined in Section 4.2.1 below) we saw and the list of the papers we saw it in.

4.2.1 Papers Containing Examples

These 101 papers contain 302 specifications, with a mean of 3 and a median of 2.

22% of these specifications are accumulation typestate systems. However, there is a significant amount of duplication between the papers in this dataset – many papers use the same few examples of typestate automata to motivate their general work on typestate.

We de-duplicated the typestate automata in these papers by combining instances of the same automaton into a single *typestate problem*: for example, we counted every one of the 19 papers that we observed using the classic `File` example (Figure 1) as a single instance of the `File` typestate problem. Considering problems rather than specifications, we found that these 101 papers only contain 114 problems. Of those 114, 31 are accumulation typestate problems (27%), indicating that there is slightly more duplication among the non-accumulation typestate specifications. Perhaps this is because papers dealing with general typestate analysis want to motivate their use of an alias analysis – which requires at least one non-accumulation typestate example. We discuss this discrepancy further in Section 4.3.

Next, we give the three most common examples of typestate problems that are accumulation and are not accumulation typestate systems.

4.2.1.1 Examples of Typestate Problems That Are Accumulation

The problem of detecting resource leaks (Figure 6) appears 16 times across 14 papers[10] [17, 39, 72, 37, 64, 42, 43, 13, 21, 19, 3, 1, 63, 51]. This problem was already known to be accumulation [37].

The need to call a distinguished initialization method on an object after its constructor finishes but before using it appears 7 times across 4 papers [24, 17, 57, 69]. For example, when using a `Socket` object, one must call `connect()` before using it to send data (Figure 7).

A third common accumulation problem is that of object initialization: before an object is fully constructed, all of its logically-required fields must be set to reasonable values (Figure 8). This pattern appears 6 times across 6 papers [35, 36, 54, 21, 27, 30]. A variant of this problem

[9] `https://doi.org/10.5281/zenodo.5771196`

[10] We tried to stay as true as possible to the story each paper presented, which is why some automata appear multiple times in the same paper. The paper treated them differently, but we believe them to be the same example. For instance, [17] discusses memory leaks and leaked sockets, which are both resource leaks.

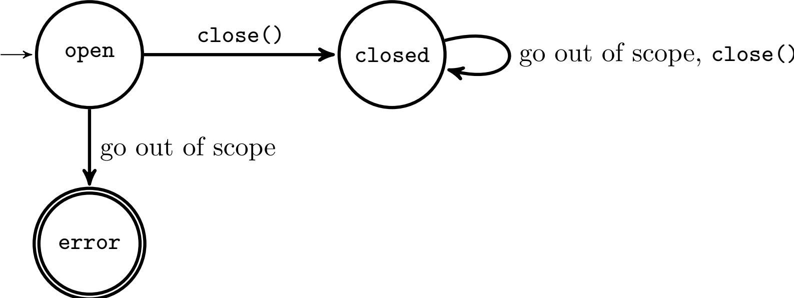


Figure 6 The typestate automaton for a resource leak, which is an accumulation typestate problem.

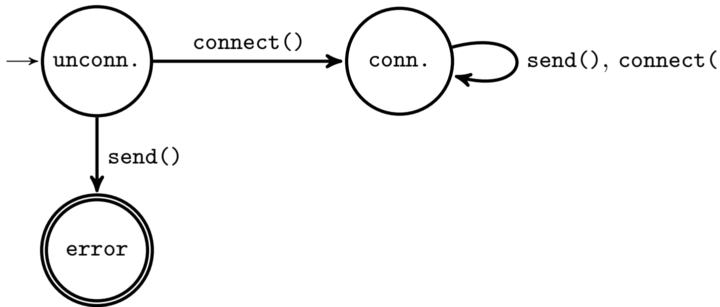


Figure 7 The typestate automaton for connecting a socket before sending data using it, which is an accumulation typestate problem.

– which arises when using the builder pattern – was known to be accumulation [35]. However, our literature survey has shown that bespoke analyses for other kinds of object initialization are also, in effect, bespoke accumulation analyses. For example, masked types [54] are a type system for ensuring that before a constructor exits, all non-null fields of the constructed class have been set to non-null values. This type system can be viewed as an accumulation analysis: the goal transition is the end of the constructor, and the enabling operations are the setting of the fields.

4.2.1.2 Examples of Typestate Problems That Are Not Accumulation

The most common non-accumulation typestate problem is "don't read or write to a stream or file after it is closed" (Figure 9), which appeared 31 times across 17 papers [24, 8, 10, 46, 25, 57, 5, 6, 53, 34, 44, 19, 71, 45, 69, 68, 11]. This problem is related to the file specification in Figure 1, but is slightly weaker – it assumes that the file is never re-opened. That this example is not accumulation demonstrates that accumulation typestate automata are a different category than automata without loops other than self-loops (a category that includes both this one and the three accumulation typestate examples in section 4.2.1.1).

"Do not update a collection while iterating over it" (Figure 10) appeared 21 times across 14 papers [9, 65, 47, 26, 51, 68, 8, 10, 33, 32, 52, 53, 7, 46]. This property is representative of an important class of properties that are never accumulation typestate systems: "disable x after y" properties that forbid the programmer from performing operation x once operation y has been performed. The key reason that these properties cannot be checked without aliasing information – and are therefore not accumulation – is that that the "disabling" operation ("start iterating" in this example) might be performed through any alias, but once it occurs, "update" must be prevented for all aliases.

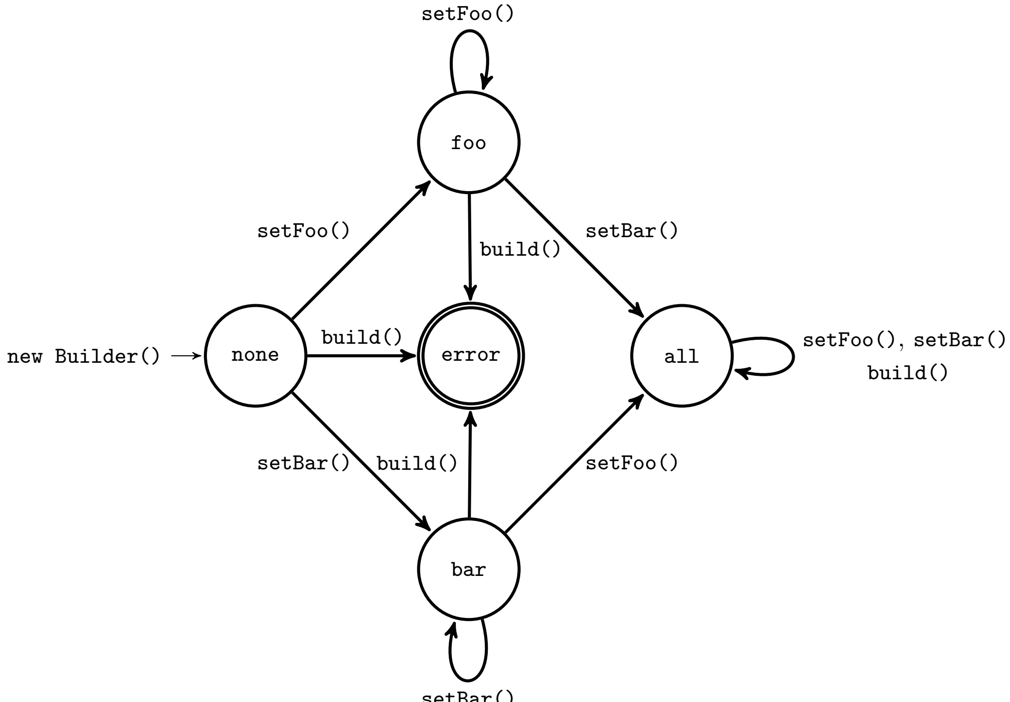


Figure 8 The typestate automaton for setting the required fields of an object before it is built, which is an accumulation typestate problem. This instance of the general pattern is specifically for a builder-pattern-style object construction pattern of a class with two required fields `foo` and `bar`.

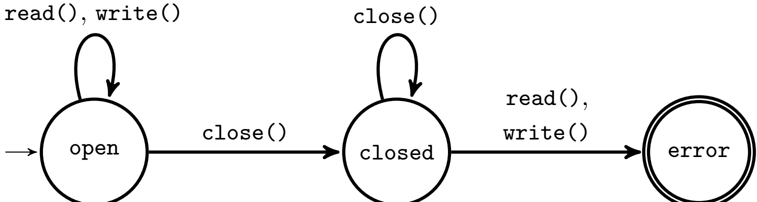


Figure 9 The typestate automaton for not reading or writing a stream after it has been closed, which is not an accumulation typestate problem.

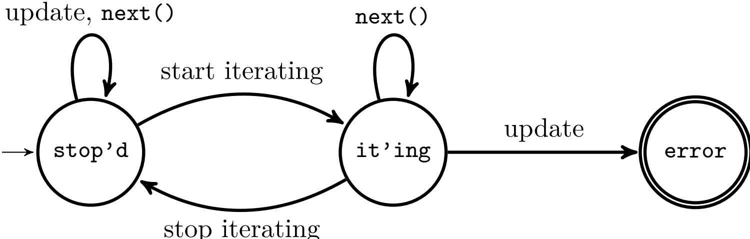


Figure 10 The typestate automaton for not updating a collection during iteration, which is not an accumulation typestate problem. Note that this automaton includes operations that are not method calls (e.g., "start iterating", which can refer to a `while` loop, a `for` loop, a map or filter operation, etc.).

The classic full file specification (Figure 1) appeared 20 times across 19 papers [28, 70, 59, 25, 29, 62, 67, 57, 69, 66, 1, 49, 16, 38, 2, 15, 72, 19, 20]. Some parts of this specification could be enforced with an accumulation analysis if a slightly different design had been chosen for the API. In particular, if files could not be re-opened once they had been closed, enforcing "only call close after open" and "only call read after open" would become accumulation properties. Since most programmers usually create a new `File` object rather than re-using an existing one, this restriction would not be particularly burdensome, but would enable easier analysis.

4.2.2 Papers With Many Typestates

This section discusses two papers that report on large collections of typestate automata.

4.2.2.1 Patterns in Property Specifications for Finite-State Verification

The first paper reports on 555 typestate-like specifications collected from a survey of 34 papers from the scientific literature, verification tool authors, and students in 1999 [18]. These 555 specifications were not de-duplicated. This paper inspired us to conduct the updated survey in Section 4.2.1. Because it precedes the start date for our survey, it is not included in the 187 papers in Section 4.2.1. We include its data here for completeness, and to discuss the differences between their results and ours (Section 4.3).

The primary goal of the paper was to categorize "finite-state properties" – that is, those expressible as finite-state machines – into patterns to help users of verification tools that take an FSM as input (such as typestate verifiers) create their own specifications by instantiating existing patterns. They categorized 511 of the 555 specifications into eight "patterns." Our analysis of these patterns is that instances of 5 of the 8 are always accumulation typestate systems (Existence, Precedence, Chain Precedence, Response, Chain Response), and some instances of a 6th (Bounded Existence, when the property is "at least" rather than "exactly" or "at most") are, as well. The 5 "always accumulation" patterns account for 306 of the 511 specifications that were categorized (60%).

4.2.2.2 An Empirical Study of Object Protocols in the Wild

The second paper [4] studies the object protocols – that is, the behavioral specifications – of all classes in four large, open-source Java projects (one of which is the Java standard library). They also categorized these specifications based on common characteristics, much like the previous study, but they created their own set of categories.

The found 648 object protocols, which were not de-duplicated. We exclude their "type qualifier" category (106 specifications), which contains classes that behave as one of a fixed set of subtypes and can never change state. The remaining 542 protocols are typestate specifications.

Instances of their most common category, Initialization, are always accumulation typestate specifications. This category contains 182 of the 542 protocols (34%). The other 6 categories (66%) are not accumulation.

4.3 Discussion

Both of the papers that reported on large sets of typestate properties included larger proportions of accumulation properties than our literature survey found otherwise. One possible explanation is that papers on novel analysis techniques tend to include "exciting" or

"challenging" problems – and, in the case of general typestate analysis, those problems usually involve aliasing (perhaps to justify the need for an alias analysis when analyzing an arbitrary typestate system, as we do in Section 1 in reference to Figure 1). Another possible explanation is that neither of the papers that reported on large sets of specifications de-duplicated their specifications, so maybe they contain many duplicate accumulation properties. When we de-duplicated the specifications in Section 4.2.1, we found that non-accumulation typestate properties tended to be duplicated more often than accumulation typestate properties. This suggests that our results may be understating the prevalence of accumulation properties. If our results understate how common accumulation properties are in practice, that is good news for practitioners interested in applying verification: we have shown that accumulation properties are easier to check than general typestate properties.

Beckman et al. [4] is the most relevant to practical programmers interested in deploying accumulation analysis. A promising avenue of future work would be a similar study to Beckman et al.'s [4] (section 4.2.2.2) on a larger corpus of software combined with automation of our decision procedure for checking whether a typestate specification is accumulation, which would permit a more reliable estimate of the percentage of typestate specifications that appear in practice that are accumulation.

Another observation is the relationship between different typestate specifications of the same type. For example, three of the examples we gave in Section 4.2.1 are applicable to `File` objects: resource leaks (Figure 6), the classic file specification (Figure 1), and reading/writing a closed file (Figure 9). Enforcing all these properties with a single typestate analysis would necessarily require alias analysis, but enforcing just the resource leak property does not – and the same might be true of other partial specifications, such as "only call read after open" – especially if files cannot be re-opened after being closed. We suspect this may be a reason why prior work did not identify a category equivalent to accumulation: many accumulation properties are sub-properties of the full typestate specification of the relevant type. That said, accumulation properties are often interesting on their own – resource leaks, for example, are harder to detect dynamically than most other types of misuses of files – and we have shown that they are easier to enforce statically.

5 Practicality of Accumulation Analysis

We implemented a general accumulation checker for Java using the Checker Framework [48] and have made it publicly available.[11] We have re-implemented the bespoke "accumulation for the builder pattern" analysis from our prior work [35] on top of it, and our "accumulation for resource leaks" analysis [37] used the general infrastructure from its inception. An accumulation analysis could be implemented modularly using any sound program analysis technique: dataflow analysis, abstract interpretation, type systems, etc. We chose a type system for convenience, and because types are naturally modular: type annotations on procedure boundaries and fields act as summaries, and local type inference infers operations that may have occurred within each procedure. Our implementation tracks enabling sets rather than enabling sequences (see Section 3.4).

We tested our implementations on the test suites of the bespoke analyses from our prior work and on the case studies that those papers describe, and found that the implementations using the common framework produced the expected results. The test suites contain both

[11] `https://checkerframework.org/manual/#accumulation-checker`

positive examples (i.e., expected errors) and negative examples (i.e., safe code). The test suites consist of 153 source files comprising 5,452 lines of non-comment, non-blank Java code. The case studies together comprise 635,006 lines of non-comment, non-blank Java code.

Our prior work also demonstrates the utility and practicality of accumulation analyses (see Section 6.1). Here are some examples from prior work:

- An accumulation analysis for verifying the absence of an initialization-related security vulnerability had 100% recall (as this paper proves, the accumulation analysis was sound!) and 82% precision – 16 true bugs vs. 3 false positives – in 9 million non-comment, non-blank lines of Java code (table 1 of [35]).
- An accumulation analysis for verifying the absence of resource leaks had 100% recall and 26% precision on 3 pieces of distributed-systems infrastructure used as a benchmark by prior work (table 4 of [37]). This compares favorably to the 13% recall and 25% precision achieved by an unsound heuristic bug-finder and the 7% recall and 50% precision achieved by a state-of-the-art typestate-based analysis that uses a (very slow) whole-program alias analysis. This precision might seem disappointing for a bug-finding tool, but we think it is acceptable for a verification tool – especially for an important and difficult problem such as resource leaks.

If the low precision of 26% for resource leaks is primarily due to lack of whole-program alias analysis – that is, if precision is much higher with comprehensive aliasing information – then there might be little point in running an accumulation analysis: it might be better to run a slow standard typestate analysis and reduce the human effort to examine false positives. This is not the case, however. We examined each false positive in [37] to determine its cause. Even with a hypothetical alias analysis that can reason precisely and flow-sensitively about the contents of collection data structures like lists or maps (which is known to be very challenging), the typestate analysis would achieve only 34% precision. A more realistic state-of-the-art (and still slow) alias analysis would give less than half of that benefit. Proving the absence of resource leaks is a difficult problem, and aliasing is not the only complication – other significant causes of false positives included bugs in the underlying analysis platform, the need to reason about nullness, and the need to reason about boolean logic.

5.1 Aliasing in Practical Accumulation Analyses

A benefit of the accumulation analysis approach is that the core accumulation analysis (Definition 4) is sound even without any alias reasoning, by Corollary 10. But it is easy to utilize aliasing information that is readily available (or cheap to compute) to improve precision. In practice, using some aliasing information is necessary to achieve acceptable precision, and untracked aliasing is usually the single biggest cause of remaining false positives even after acceptable precision has been achieved.

Our prior work [35, 37] used cheap, targeted must-alias reasoning to improve the precision – that is, the false positive rate – of the analyses. For example, section 4.3 of [35] and sections 3–5 of [37] give lightweight aliasing analyses. These lightweight alias analyses compute only the aliasing information necessary to remove false positives that occurred in practice for these analyses, which makes them much cheaper than computing precise aliasing information for all variables (of types with typestate automata) in the program, as a whole-program alias analysis would.

Our general accumulation checker includes both the suite of built-in cheap sound must-alias analyses from prior work and hooks for analysis developers to add further aliasing information.

5.2 Handling Other Features of Real Programming Languages

The core calculus in section 3.3.1.1 does not model features that are present in a practical programming language, including unanalyzed dependencies, open programs, class definitions, conditionals, inheritance, etc. Our formalism already handles some of these: for example, handling conditionals requires a may-aliasing oracle and estimated sets of typestates rather than a single typestate, both of which our formalism includes. Extending our proofs to other features is straightforward and does not require new proof techniques.

An advantage of accumulation analysis is that in practice it is possible to soundly handle code with unknown or "arbitrarily-bad" effects – including unmodeled features of the target language – by reverting to a safe default, in the same manner as an abstract interpretation might "go to top" in the presence of side effects. For example, if a call to an un-analyzed method might re-assign a field, an accumulation analysis can conservatively assume that that field's value is in the typestate automaton's start state after the call. This is sound as a consequence of Lemma 15 and the definition of accumulation (in the same manner as Lemma 16): the start state is necessarily a sound default assumption, because all goal transitions must be forbidden in it.

By contrast, in a non-accumulation typestate system it is not sound to fall back to the automaton's start state. For example, consider the `File` example in Figure 1: the start state is `closed`, where `open()` is a legal call. But treating all field reads as returning `closed` files would not be sound, because if the underlying `File` value was actually in the `open` state, a sound analysis should issue an error for a subsequent call to `open()`.

An advantage of our choice of a pluggable type system to implement our accumulation analyses is that the "start state" of a field can be changed by changing its declared type to specify a different typestate. This restricts that field to only contain values whose typestates are in the states reachable from the declared typestate – that is, the sub-automaton composed of states reachable from the declared type. For the accumulation analyses we implemented, we found that this ability to refine a field's declared type to be sufficient to enable precise analysis of field reads.

6 Related Work

6.1 Previous Work on Accumulation

Our prior work [35, 37] uses accumulation analyses to solve specific typestate-like problems (object initialization via the builder pattern and resource leak prevention). One of these [35] gives an informal relationship between accumulation and typestate: we claimed that a typestate automaton can be checked with an accumulation analysis if "(1) the order in which operations are performed does not affect what is subsequently legal, and (2) the accumulation does not add restrictions; that is, as more operations are performed, more operations become legal." We did not substantiate this definition with a proof, and it is not quite equivalent to the definition of an accumulation typestate system we use in this paper, which does permit some kinds of ordering properties (see Section 3.4). This paper makes more precise claims and provides a proof that the analyses are sound (Corollary 10).

6.2 Heap Monotonic Typestates

Heap-monotonic typestates [22] are a class of typestate that, like accumulation typestate systems, do not require aliasing information for soundness. A heap monotonic typestate system is one in which the statically observable invariants of the relevant type become monotonically stronger as an object transitions through its typestates. Every heap-monotonic typestate system is an accumulation typestate system.

The present work goes further than the work on heap-monotonic typestates in three important ways. First, we have shown exactly which typestate systems (the accumulation typestate systems) can be checked without aliasing; heap-monotonic typestate systems were proven to be sound without aliasing information, but not proven to encompass all typestate systems that can be soundly checked without aliasing. Second, we have surveyed the literature to locate examples of typestate systems that can be checked soundly without aliasing; the paper on heap-monotonic typestates gives a few examples, but no procedure for discovering more. Third, we have implemented practical accumulation analyses: the prior work on heap-monotonic typestates was, to the best of our knowledge, entirely theoretical.

6.3 Other Categories of Typestate Systems

Others have identified interesting sub-categories of typestate systems that may be amenable to different kinds of analysis. While as far as we are aware we are the first to identify the accumulation typestate systems, the omission-closed typestate systems [23] are a close relative. An omission-closed typestate system is one in which every subsequence of every valid (i.e., not ending in the `error` state) path is also a valid path. In other words, omission-closed properties are those whose *valid* paths are closed under subsequence. By contrast, accumulation typestate systems are those whose *error-inducing* paths are closed under subsequence, if the last error-inducing transition is held constant. Unlike accumulation typestate systems, not all omission-closed typestate systems can be checked soundly without aliasing: for example, the typestate system for a `File` object whose FSM is defined by the regular expression “read*;close” is omission-closed, but cannot be checked soundly without aliasing information, because it is an error to call “close” more than once – or, put another way, “close” disables itself. Omission-closed typestate properties are of interest because they can be verified in polynomial time for *shallow* programs – programs where all pointers are “single-level”: that is, where no pointer refers to a value that itself contains a pointer.

6.4 Typestate Surveys

Section 4.2.2 describes two previous papers that report on large quantities of typestate specifications [4, 18]. We have extended their work by surveying 101 papers that neither of those works considered and locating all typestates within them, and by identifying which typestate systems are accumulation typestate systems.

6.5 Practical Typestate Analyses

There have been many attempts to improve the scalability of typestate analyses. We mention only some of the most recent here. Rapid [21] is a modern typestate analysis built at AWS. Rapid’s scalability is a design choice: it is intentionally unsound and therefore scales by not tracking all aliasing. Another recent example is Grapple [72], which uses a novel graph-reachability algorithm and a modern alias analysis together. Some of Grapple’s optimizations make it unsound despite access to aliasing information. Because Grapple does track aliasing, it scales much poorly than accumulation-based systems: for example, Grapple is more than an order of magnitude slower than an accumulation-based approach to resource-leak detection [37].

6.6 Typestate With Aliasing Restrictions

Another method to avoid the need to do an expensive whole-program alias analysis is to limit the programmer's use of aliasing. Examples include linear or affine type systems [16, 61], role analysis [40], ownership types [14, 55], and access permissions [7]. Accumulation analyses, unlike all of these approaches, do not impose any restrictions on the programming model.

6.7 Other Work on Typestate

Typestate is well-studied in the scientific literature, and there is not space to give a full survey here. However, our artifact[12] mentions all the papers that we examined as part of our literature survey (Section 4).

7 Conclusion

Soundly checking an accumulation typestate system is significantly cheaper than soundly checking an arbitrary typestate system because it is not necessary to compute exhaustive aliasing information. Since the expense of computing exhaustive aliasing information has been a key barrier for the adoption of sound typestate analyses in practice, we believe that accumulation analysis is a promising approach for the estimated 41% (Table 1) of typestate specifications that are actually accumulation typestate specifications. Typestate analysis designers or users can use our work to check whether their specification is an accumulation typestate specification, and if it is, they can use an accumulation analysis – gaining an order of magnitude or more in analysis speed at only a small cost in precision.

References

1 Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *International Static Analysis Symposium*, pages 230–246. Springer, 2002.

2 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022, 2009.

3 Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 143–162, 2008.

4 Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer, 2011.

5 Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. *ACM SIGSOFT Software Engineering Notes*, 30(5):217–226, 2005.

6 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *ACM SIGPLAN Notices*, 42(10):301–320, 2007.

7 Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, pages 195–219. Springer, 2009.

8 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 5–14. IEEE, 2010.

[12] `https://doi.org/10.5281/zenodo.5771196`

9 Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming*, pages 525–549. Springer, 2007.

10 Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47, 2008.

11 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250. IEEE, 2011.

12 Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, 2009.

13 Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 480–491, 2007.

14 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, Berlin, Heidelberg, 2013.

15 Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, 2002.

16 Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with Java(X). In *European Conference on Object-Oriented Programming*, pages 550–574. Springer, 2007.

17 Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, 2001.

18 Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, pages 411–420, 1999.

19 Matthew B. Dwyer, Madeline Diep, and Sebastian Elbaum. Reducing the cost of path property monitoring through sampling. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 228–237. IEEE, 2008.

20 Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 124–133, 2007.

21 Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. RAPID: Checking API usage for the cloud in the cloud. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1416–1426, 2021.

22 Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, pages 58–72, Darmstadt, Germany, July 2003.

23 John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. In *International Static Analysis Symposium*, pages 439–462. Springer, 2003.

24 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008.

25 Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, 2002.

26 Asya Frumkin, Yotam M. Y. Feldman, Ondřej Lhoták, Oded Padon, Mooly Sagiv, and Sharon Shoham. Property directed reachability for proving absence of concurrent modification errors. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 209–227. Springer, 2017.

27 Mark Gabel and Zhendong Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

28 Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. *ACM Sigplan Notices*, 46(3):239–250, 2011.

29 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–44, 2014.

30 Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *European Conference on Object-Oriented Programming*, pages 520–545. Springer, 2009.

31 Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336, 1952.

32 Jeff Huang, Qingzhou Luo, and Grigore Rosu. GPredict: Generic predictive concurrency analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 847–857. IEEE, 2015.

33 Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Rosu. Garbage collection for monitoring parametric properties. *ACM SIGPLAN Notices*, 46(6):415–424, 2011.

34 Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded Java programs. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 288–296. IEEE, 2008.

35 Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. Verifying object construction. In *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*, pages 1447–1458, Seoul, Korea, May 2020.

36 Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. Continuous compliance. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, pages 511–523, Melbourne, Australia, September 2020.

37 Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and modular resource leak verification. In *ESEC/FSE 2021: The ACM 29th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Athens, Greece, August 2021.

38 Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 416–428, 2009.

39 Goh Kondoh and Tamiya Onodera. Finding bugs in Java Native Interface programs. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 109–118, 2008.

40 Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–32, 2002.

41 William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL '91: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.

42 Wei Le and Mary Lou Soffa. Marple: Detecting faults in path segments using automatically generated analyses. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):1–38, 2013.

43 Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Memfix: static analysis-based repair of memory deallocation errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 95–106, 2018.

44 Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

45 Filipe Militao, Jonathan Aldrich, and Luís Caires. Rely-guarantee protocols. In *European Conference on Object-Oriented Programming*, pages 334–359. Springer, 2014.

46 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. *ACM Sigplan Notices*, 43(10):347–366, 2008.

47 Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates in the presence of inter-object references. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 77–96, 2005.

48 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 2008.

49 Hila Peleg, Sharon Shoham, Eran Yahav, and Hongseok Yang. Symbolic automata for static specification mining. In *International Static Analysis Symposium*, pages 63–83. Springer, 2013.

50 Goran Piskachev, Tobias Petrasch, Johannes Späth, and Eric Bodden. AuthCheck: Program-state analysis for access-control vulnerabilities. In *International Symposium on Formal Methods*, pages 557–572. Springer, 2019.

51 Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 925–935. IEEE, 2012.

52 Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 270–285, 2010.

53 Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 280–290, 2013.

54 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. *ACM SIGPLAN Notices*, 44(1):53–65, 2009.

55 Rust team. Rust programming language. `https://www.rust-lang.org/`, 2021. Accessed 30 November 2021.

56 Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

57 Johannes Späth, Karim Ali, and Eric Bodden. IDEal: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.

58 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, SE-12(1):157–171, January 1986.

59 Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In *OOPSLA 2011, Object-Oriented Programming Systems, Languages, and Applications*, pages 713–732, Portland, OR, USA, October 2011.

60 Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

61 Jesse A. Tov and Riccardo Pucella. Practical affine types. *ACM SIGPLAN Notices*, 46(1):447–458, 2011.

62 Cláudio Vasconcelos and António Ravara. From object-oriented code with assertions to behavioural types. In *Proceedings of the Symposium on Applied Computing*, pages 1492–1497, 2017.

63 Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 185–195, 2016.

64 Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, 2016.

65 Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. Arc++: effective typestate and lifetime dependency analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 116–126, 2014.

66 Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–50, 2014.

67 Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–234, 2008.

68 Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. Symbolic verification of regular properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 871–881. IEEE, 2018.

69 Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 1–12, 2014.

70 Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. *ACM SIGPLAN Notices*, 48(6):365–376, 2013.

71 Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. Regular property guided dynamic symbolic execution. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 643–653. IEEE, 2015.

72 Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *EuroSys*, pages 1–17, 2019.

A Proof of Lemma 15

This appendix contains the full proof of Lemma 15, which appears in section 3.3.1.6 and is used by Lemma 16, the forwards direction of the proof of Theorem 9. We begin by restating Lemma 15:

▶ **Lemma 15.** *Let $R = \phi_s(x_i)$ be the set of estimated typestates produced by a typestate analysis with no aliasing information for a variable x_i before a statement s. Let S be the trace of an arbitrary execution leading up to some occurrence of s, and let $t = \tau(\rho(x_i))$ be the typestate of the actual value to which x_i refers before that occurrence of s. Applying S to the automaton leads to typestate t. There exists a typestate $r \in R$ such that applying some subsequence of S leads to r. That is, there is some estimated typestate $r \in R$ that is reachable by a subsequence of the transitions that lead to t.*

The proof is by co-induction on the dynamic semantics of the language in Figure 2 and the definition of a typestate analysis with no aliasing information in Definition 14, with one change to its rule for load operations (rule TS-LOAD-FIX in Figure 5). In particular, the load rule our typestate analysis with no aliasing uses in this proof is the following:

- For a load statement s, where s is x_i := x_j.f_k, let s_0 be the start state of the automaton A which is being checked. The analysis updates its estimate for x_i so that it is mapped to s_0: $\phi'_s(x_i) := s_o$. For all other names m in ϕ_s where $m \neq x_i$, the analysis copies the entry from the previous abstract store: $\phi'_S(m) := \phi_s(m)$.

(See the discussion of why this modified rule is necessary in section 3.3.1.6, after the original statement of Lemma 15.)

Proof.

Base case: when a program begins executing, the dynamic semantics say that all names refer to values in the start state. A typestate analysis with no aliasing information estimates that at a program's entry point, all names are in the start state, as well. Trivially, the start state is reachable by the same sequence of operations as itself.

Case assignment: For an assignment s, where s is x_i := x_j, the invariant is preserved by the inductive hypothesis. Consider that by the inductive hypothesis, the invariant is preserved for x_j. Then consider the rule used by the typestate analysis with no aliasing information for an assignment: every mention of x_i in the abstract store is replaced by x_j. Further, the dynamic semantics for an assignment require that the previous value of x_i is no longer accessible via x_i: x_i after the assignment refers only to x_j. Since x_i and x_j after the assignment are treated entirely the same, but the abstract store is otherwise unchanged by the analysis, what was true of x_j before the statement is true for x_i after.

Case load: The special load rule TS-LOAD-FIX trivially guarantees that the invariant is preserved: the start state is reachable by a subsequence of the operations that reach any other state (in particular, by the empty subsequence).

Case store: This rule trivially preserves the invariant, because the invariant must be maintained only for the estimates for variables – not for fields – and rule TS-STORE only updates estimates for fields.

Case method call: For a method call $s = x_i$.m_j(), only steps 1 and 2 of rule TS-CALL are applied, because a typestate analysis with no aliasing information never performs strong or weak updates on possible aliases. The invariant is preserved via the inductive hypothesis: for x_i itself, let r_1 be the element of R that is reachable by a subsequence of the actual sequence S in the inductive hypothesis. The analysis updates its estimate to include $r_1 + m_j$ (that is, the sequence r_1 followed by the transition m_j). After s is executed, the actual sequence is $S + m_j$, and since we know that r_1 is reachable by a subsequence of S, $r_1 + m_j$ must be reachable by a subsequence of $S + m_j$ – the same subsequence used to reach r_1, with m_j added on. For any aliases of x_i, the inductive hypothesis also guarantees that the invariant holds: the estimate contains some r that is a subsequence of S, and any subsequence of S is also a subsequence of $S + m_j$.

Case sequence: For a sequence, the invariant is trivially preserved by induction. ◀

Concolic Execution for WebAssembly

Filipe Marques
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

José Fragoso Santos
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Nuno Santos
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Pedro Adão
Instituto Superior Técnico, University of Lisbon, Portugal
Instituto de Telecomunicações, Aveiro, Portugal

Abstract

WebAssembly (Wasm) is a new binary instruction format that allows targeted compiled code written in high-level languages to be executed by the browser's JavaScript engine with near-native speed. Despite its clear performance advantages, Wasm opens up the opportunity for bugs or security vulnerabilities to be introduced into Web programs, as pre-existing issues in programs written in unsafe languages can be transferred down to cross-compiled binaries. The source code of such binaries is frequently unavailable for static analysis, creating the demand for tools that can directly tackle Wasm code. Despite this potentially security-critical situation, there is still a noticeable lack of tool support for analysing Wasm binaries. We present WASP, a symbolic execution engine for testing Wasm modules, which works directly on Wasm code and was built on top of a standard-compliant Wasm reference implementation. WASP was thoroughly evaluated: it was used to symbolically test a generic data-structure library for C and the Amazon Encryption SDK for C, demonstrating that it can find bugs and generate high-coverage testing inputs for real-world C applications; and was further tested against the Test-Comp benchmark, obtaining results comparable to well-established symbolic execution and testing tools for C.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Security and privacy → Formal methods and theory of security

Keywords and phrases Concolic Testing, WebAssembly, Test-Generation, Testing C Programs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.11

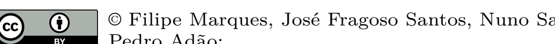
Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.20`

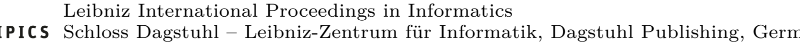
Funding The authors were supported by national funds through Fundação para a Ciência e a Tecnologia (UIDB/50008/2020, Instituto de Telecomunicações, and UIDB/50021/2020, INESC-ID multi-annual funding), projects INFOCOS (PTDC/CCI-COM/32378/2017) and DIVINA (CMU/TIC/0053/2021), and by the European Commission under grant agreement number 830892 (SPARTA).

Acknowledgements We are grateful to Carolina Costa with whom we designed a preliminary version of WASP and its concolic semantics as part of her M.Sc. thesis [22].

1 Introduction

WebAssembly (Wasm) [30] is a binary instruction format designed to be the new standard compilation target for the Web and is now supported by all major browser vendors, enabling Web applications to run with near-native speed. As a result, Web applications are increasingly being ported into Wasm to reap its performance benefits. In particular, Wasm has been adopted in server-side runtimes [13, 1, 20], IoT platforms [31], and in edge computing [34].


© Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão;
licensed under Creative Commons License CC-BY 4.0



The compilation of unsafe languages to Wasm opens up the opportunity for the introduction of new classes of bugs and security vulnerabilities into the setting of Web programs, as such issues in the original programs can be transposed to Wasm binaries via compilation [23]. This is the case, for example, of buffer overflows [50], format string bugs [9], and use-after-free errors [29]. By exploiting such flaws [25], cyber attackers have access to a widened surface for launching their attacks on the Web. These include cross-site scripting attacks by exploiting client-side code [50], or code injection attacks by targeting vulnerabilities in server-side code (e.g., powered by Node.js). In addition, Wasm itself can be used for writing malware, e.g., web keyloggers [49], or crypto-miners [59]. Importantly, Wasm binaries are often integrated directly into Web applications, with developers not having access to the corresponding source code. In this scenario, developers must analyse stand-alone Wasm code to test it against potential security vulnerabilities and other types of execution errors.

Symbolic execution [15, 5] is a program analysis technique that allows for the exploration of multiple program paths by running the given program using symbolic values instead of concrete ones. It has successfully been applied to finding a wide range of security vulnerabilities and other types of bugs in many high-level and intermediate languages, including C [14, 28], Java [54], and JavaScript [61, 60, 26]. Nonetheless, to the best of our knowledge, there are only two tools for symbolically executing Wasm code: WANA [73] and Manticore [51]. Both these tools are, however, mainly targeted at the analysis of smart contracts and have important limitations that constraint their application to stand-alone Wasm modules. WANA [73] is in preliminary development stages and can only be applied to EOSIO and Ethereum smart contracts, since it does not include a symbolic execution engine for Wasm that can be run on its own. Manticore [51] has recently gained support for Wasm [33], but has not yet been systematically evaluated on Wasm code. Furthermore, its application to Wasm modules requires the manual setup of a complex Python script for each possible input memory, making it cumbersome and difficult to automate.

We present the WebAssembly Symbolic Processor, WASP, a novel concolic execution engine for testing Wasm (version 1.0) modules. WASP follows the so-called *concolic discipline* [28, 64], combining concrete execution with symbolic execution and exploring one execution path at a time. However, unlike most concolic execution engines [65, 79, 64, 63], which are implemented via program instrumentation, we implement WASP by instrumenting the Wasm reference interpreter developed by Haas et al. [30]. To this end, we lift the authors' reference interpreter from concrete values to pairs of concrete and symbolic values. By moving the instrumentation to the interpreter level, we open up the possibility for a range of optimisations in the context of concolic execution. In this paper, we explore two such optimisations: **(1)** application of algebraic simplifications to byte-level symbolic expressions generated by memory interactions (§3.3); and **(2)** shortcut restarts for failed assumption statements (§3.4). Finally, we formalise our concolic analysis as a small-step concolic semantics, which we use to both guide the implementation of WASP and describe its mathematical underpinnings. This semantics is an additional contribution of the paper as we are not aware of any such formalisation of concolic execution for a low-level language.

While our first goal is for WASP to be able to analyse stand-alone Wasm modules, we also aim for it to be used as a common platform for building symbolic analyses for high-level programming languages that compile to Wasm. In a nutshell, if one wants to use WASP to enable a symbolic execution engine for a given language, one has to accomplish the following two tasks. First, the symbolic primitives of WASP, such as the declaration of assertions, assumptions, and the creation of symbolic variables, must be exposed at the source-language level and properly connected to the corresponding WASP primitives via compilation. Second,

one must guarantee that either the code of the required runtime libraries is available for symbolic execution or that WASP includes symbolic summaries that model the behaviour of those libraries. In order to demonstrate the viability of this approach, we use it to build WASP-C, a new symbolic execution framework for testing C programs. WASP-C shows that, with a relatively small effort ($\approx$ 800 LOC), we were able to build a new concolic engine for C that is able to analyse industry-grade code and obtain results comparable to well-established symbolic execution and testing tools for C, such as KLEE [14] and VeriFuzz [7].

We evaluate WASP in the five following ways: **(1)** We compare the performance of WASP with that of Manticore [51] in the analysis of a stand-alone Wasm B-tree data structure [22]. We demonstrate that WASP outperforms Manticore, its only competitor tool. **(2)** We use WASP-C to symbolically test Collections-C [52], a widely-used generic data structure library for C previously tested using the Gillian-C tool [26]. WASP-C found three bugs during the testing process, including a previously unknown bug that Gillian-C did not detect. Also, WASP-C is more efficient than Gillian-C, completing the symbolic analysis of the library 14% faster. **(3)** We run WASP-C against the Test-Comp [10] benchmark, obtaining results comparable to well-established symbolic execution and testing tools for C, such as KLEE [14] and VeriFuzz [7]. If we compare the results we obtained for WASP-C against those obtained for the tools submitted for the 2021 Competition on Software Testing (TestComp 2021 [11]), we conclude that WASP-C is the third-best tool in the *cover-error* category and the sixth-best tool in the *cover-branches* category out of a total of twelve tools (with WASP included). **(4)** We measure the impact of the proposed optimisation techniques on the performance of WASP by comparing the execution times obtained for WASP with the optimisations enabled against those obtained with them disabled. Results indicate the proposed optimisations are essential for WASP's performance. **(5)** We use WASP-C to symbolically test the Amazon Encryption SDK [67], generating a high-coverage test suite for that library and demonstrating that WASP-C scales to industry-grade code.

Our evaluation is mostly focused on WASP-C due to the fact that there is no symbolic benchmark for stand-alone Wasm modules. However, according to a recent study [35], most Wasm code on the Web ($\approx$ 65%) comes from C/C++ applications. In this light, we believe it is appropriate to center the analysis of the performance of WASP on compiled C code.

Contributions. In summary, the contributions of this work are three-fold: **(1)** WASP, a concolic execution engine for Wasm (§3); **(2)** WASP-C, a symbolic execution framework for testing C programs built on top of WASP (§3.5); and **(3)** three symbolic datasets for evaluating symbolic execution tools for Wasm, covering different types of symbolic reasoning (§4).

2 Background

In this section we give an overview of Wasm, focusing on its syntax and semantics, together with a high-level introduction to symbolic execution with a particular emphasis on the concolic approach followed in this paper.

2.1 WebAssembly

WebAssembly [30, 57] is a low-level bytecode format that offers compact representation, efficient validation and compilation, and ensures safe execution with minimal overhead. Wasm is not tied up to any specific hardware, being language-, hardware- and platform-independent. Like other assembly languages, it is mainly used as a compilation target for high-level languages, such as C/C++ or Rust, allowing for code written in a range of languages to be run on web browsers with significant speed improvements compared to JavaScript [78].

$$
\begin{array}{ll}
\textsc{Value Types} & \textsc{Function Types} \\
t ::= i32 \mid i64 \mid f32 \mid f64 & tf ::= t^* \rightarrow t^*
\end{array}
$$

$$
\begin{array}{l}
\textsc{Instructions} \\
e ::= \texttt{unreachable} \mid \texttt{nop} \mid \texttt{drop} \mid \texttt{select} \mid t.\texttt{const} \mid t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid \\
\quad t.cvtop\ t_sx^? \mid \{\texttt{get}|\texttt{set}|\texttt{tee}\}\texttt{_local}\ i \mid \{\texttt{get}|\texttt{set}\}\texttt{_global}\ i \mid t.\texttt{load}\ (\texttt{tp_sx}^?)\ a\ o \mid \\
\quad t.\texttt{store}\ \texttt{tp}^?\ a\ o \mid \texttt{current_memory} \mid \texttt{grow_memory} \mid \texttt{block}\ tf\ e^*\ \texttt{end} \mid \\
\quad \texttt{loop}\ tf\ e^*\ \texttt{end} \mid \texttt{if}\ tf\ e^*\ \texttt{else}\ e^*\ \texttt{end} \mid \texttt{br}\ i \mid \texttt{br_if}\ i \mid \texttt{br_table}\ i^+ \mid \texttt{return} \mid \\
\quad \texttt{call}\ i \mid \texttt{call_indirect}\ tf
\end{array}
$$

$$
\begin{array}{lll}
\textsc{Imports} & \textsc{Exports} & \textsc{Functions} \\
im ::= \texttt{import}\ \text{“}name\text{” “}name\text{”} & ex ::= \texttt{export}\ \text{“}name\text{”} & f ::= ex^*\ \texttt{func}\ tf\ \texttt{local}\ t^*\ e^* \mid \\
 & & \quad\ ex^*\ \texttt{func}\ tf\ im
\end{array}
$$

Figure 1 Simplified Wasm abstract syntax, as presented in [30].

Syntax. A WebAssembly binary takes the form of a *module*. A Wasm module includes a collection of Wasm *functions*, together with the declaration of their shared global variables and the specification of the linear memory where the functions and global variables are loaded. Computation is based on a *stack machine*; Wasm instructions interact with the stack by pushing values onto the stack or popping values out of the stack. A Wasm module is executed by an *embedder*, e.g., the host JavaScript engine that defines how modules are loaded, resolves imports and exports between modules, and handles I/Os, timers, and traps.

The syntax of Wasm programs is given in Figure 1 and includes: functions f, instructions e, values c, value types t, and function types tf. Wasm has four *primitive types*, all readily available on common hardware: machine-integers and IEEE 754 floating-point numbers [36], each with a 32- and 64-bit variant. Wasm makes no distinction between signed and unsigned *integers*; instead, instructions have a *sign extension* to indicate how to interpret the generated integer values. *Wasm variables* can be either *local*, belonging to the execution context of a function, or *global*, belonging to the entire module. Wasm does not have "named" variables; instead, both local and global variables are indexed by integer values. The primary storage of a Wasm module is a large array of bytes, commonly referred to as *linear memory*. The initial memory size is fixed. However, memories can be programmatically grown. In contrast to most stack machines, Wasm has structured control flow constructs, ensuring that humans can easily interpret Wasm code and that no irreducible loops [32] are encountered. Wasm *instructions* can be divided into the following categories:

- *Stack instructions*: the instruction `drop` for popping the value at the top of the stack; the instruction `const` for pushing a value onto the stack; the unary and binary operator instructions for applying the corresponding operators to the value(s) at the top of the stack, replacing that(those) value(s) with the obtained result. Operator instructions include the standard relational, arithmetic, and boolean operators.
- *Variable instructions*: the instructions `set_local` and `set_global` for updating the values of local and global variables; the instructions `get_local` and `get_global` for retrieving the values of local and global variables, placing them at the top of the stack; and the instruction `tee_local` for setting the value of a local variable to the value at the top of the stack without removing that value from the stack.
- *Memory instructions*: the instructions `load` and `store` for loading and storing primitive values from and to memory; the instruction `grow_memory` for increasing the size of the current memory one page at a time – page size is fixed at 64 KiB; and the instruction `current_memory` for obtaining the size of the current memory.
- *Control flow instructions*: the standard control-flow instructions: `loop`, `if` and `block`, `br`, `return`, `call`; and the Wasm-specific control-flow instructions `call_indirect`, used to implement dynamic dispatch at the Wasm level, and `br_tbl`, used to implement the standard `switch` statement.

The reader is referred to [30] for a thorough account of the syntax of Wasm.

2.2 Symbolic Execution

Symbolic execution is a program analysis technique used to explore all feasible paths of a program up to a bound [41]. Instead of running a program using concrete values, symbolic execution engines run the given program with *symbolic* inputs. Every time the symbolic execution engine hits a conditional expression with a symbolic guard, the engine forks the current execution to be able to explore both branches. For each execution path, the symbolic execution engine builds a first order formula, called *path condition*, which accumulates the constraints on the symbolic inputs that direct the execution along that path. In particular, every time a conditional instruction is symbolically executed, the current path condition is extended with its guard in the "then" branch and with the negation of its guard in the "else" branch. Symbolic execution engines rely on an underlying SMT solver to check the feasibility of execution paths and the validity of the assertions supplied by the developer. An execution path is said to be feasible if it can be realised by at least one concrete path and an assertion holds at a given program point if it is implied by the path condition at that point.

Concolic Execution. Concolic execution is a special variation of symbolic execution in which one pairs up a concrete execution with a symbolic execution to avoid interactions with the underlying SMT solver by exploring one execution path at a time [28, 64]. Concolic execution engines assign concrete values to symbolic inputs and execute the given program both concretely and symbolically at the same time, following only the concrete path but constructing the path condition corresponding to that path as in pure symbolic execution. The constructed path condition is instrumental to concolic execution as it captures the conditions that must hold for the execution to take the explored path. More specifically, it can be used to generate new concrete inputs for symbolic variables that will force the exploration of a different path. To this end, one needs to negate the obtained path condition and query the underlying SMT solver for a model of the obtained formula. By keeping track of all the path conditions generated via concolic execution, the engine can enumerate all program execution paths up to a bound, with the advantage of only having to interact with the underlying constraint solver one time per explored path. Note that in purely symbolic execution, the engine must query the constraint solver every time it hits a branching point in order to determine whether or not its then- and else- branches are feasible.

Concolic Execution: Example. Let us now take a look at how concolic execution works in practice. Consider the C program given in Figure 2a. This program is annotated with a final assert statement, which is supposed to hold independently of the values of variables x and y. Hence, in order to determine whether or not the assertion always holds, one has to explore all feasible execution paths of the program, which we illustrate in Figure 2b in the form of an execution tree. In the tree, we depict in green the leaf nodes corresponding to execution paths for which the assertion holds and in red those corresponding to paths for which it does not. The final assertion does not hold for the left-most path. To see this, consider the inputs $x = 1$ and $y = 4$. These inputs cause variables a and b to be both assigned to 6, violating the final assertion. Below, we explain how these inputs can be discovered.

As there are three possible execution paths, there will be three concolic executions, each corresponding to a different execution path. In the following, we will refer to these executions as *concolic iterations*. During the *first concolic iteration*, the concrete values associated with the symbolic variables of the program are picked non-deterministically from the set of all concrete values of their corresponding types. For this example, we will assume that x and y are respectively set to 0 and 2. These inputs cause the concolic execution engine to explore the rightmost path of the execution tree, generating the final path condition: $x \leq 0$.

```c
int main() {
    int a = 4, b = 2;
    int x = symb(), y = symb();
    if (x > 0) {
        b = a + 2;
        if (x < y)
            a = (x * 2) + y;
    }
    assert(a != b);
}
```

(a) Symbolic C program.

(b) Execution tree for program in Listing 2a.

Figure 2 Concolic execution example.

Before the *second concolic iteration*, the concolic execution engine queries the underlying constraint solver for a model for the symbolic inputs that satisfies the formula $x > 0$, corresponding to the negation of the first path condition. Let us assume that the solver returns the model $x = 1$ and $y = 0$. These inputs cause the concolic execution engine to explore the middle path, generating the path condition: $(x > 0) \wedge (x \geq y)$.

Before the *third concolic iteration*, the engine queries the solver for a model for the symbolic inputs that satisfies the negation of both path conditions found so far:

$$(x > 0) \wedge ((x \leq 0) \vee (x < y)) \;\; \equiv \;\; (x > 0) \wedge (x < y)$$

Assume that the solver outputs the model $x = 1$ and $y = 2$. These inputs cause the concolic execution engine to explore the leftmost path of the execution tree. Observe that this model does not immediately trigger the assertion violation, since the final values of a and b do not coincide ($a = 4$ and $b = 6$). In order to understand how the concolic execution engine finds the model that violates the assertion, one has to consider the concolic state at the point where the assert statement is encountered, which is given below:

$$x \mapsto (1, x) \quad y \mapsto (2, y) \quad a \mapsto (4, 2 \times x + y) \quad b \mapsto (6, 6) \quad PC \equiv (x > 0) \wedge (x < y)$$

Given this concolic state, the expression $a \neq b$ evaluates to the concrete value *true* and the symbolic value $(2 \times x + y) \neq 6$. In order to establish that the assertion holds, the concolic execution engine must prove that the *symbolic expression* being asserted is implied by the current path condition; put formally:

$$(x > 0) \wedge (x < y) \Rightarrow (2 \times x + y) \neq 6$$

In order to check the validity of this implication, the concolic execution engine queries the underlying constraint solver for the satisfiability of its negation:

$$(x > 0) \wedge (x < y) \wedge (2 \times x + y) = 6$$

which is satisfied by the model $x = 1$ and $y = 4$, disproving the implication and witnessing the assertion failure.

3 WASP

This section presents our concolic execution engine for Wasm named WASP. We begin by describing the architecture of WASP (§3.1). Next, we explain the concolic semantics (§3.2) and memory model (§3.3) at the core of WASP. Then, we introduce an optimised version of the proposed concolic semantics (§3.4) and conclude with a brief overview of WASP-C (§3.5).

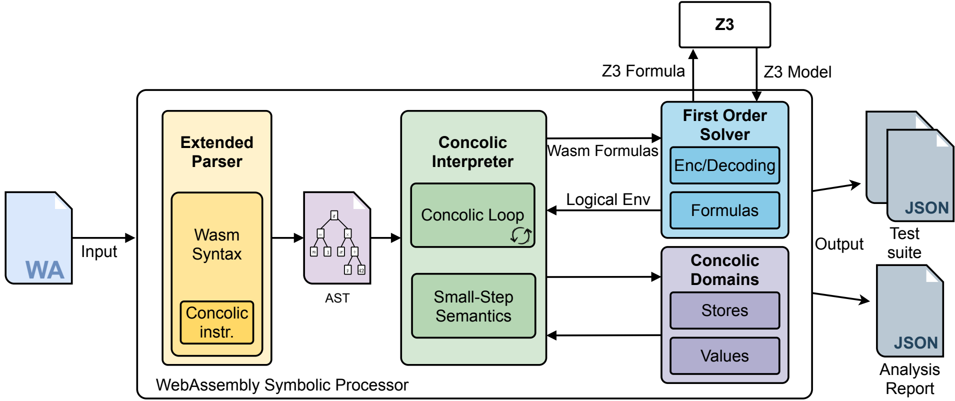


Figure 3 High-level architecture of WASP.

3.1 Overview

The goal of WASP is to explore multiple execution paths of the program to be analysed in order to uncover potential execution errors. To this end, the Wasm programs given to WASP must be annotated with first order assertions to be validated by WASP. WASP explores all the execution paths of the given program up to a pre-established depth. If no assertion failure is found, WASP provides a bounded verification guarantee. Otherwise, it outputs a concrete counter-model that triggers that failure.

WASP was developed on top of the Wasm reference interpreter [56], which we extended with symbolic facilities according to the high-level architecture described in Figure 3. In particular, we extended the original code-base of the reference interpreter with: **(1)** parsing facilities for the symbolic instructions required for declaring and reasoning over symbolic inputs; **(2)** a new concolic interpreter module, implementing the main concolic loop and the concolic execution of Wasm instructions; **(3)** a new concolic state module, implementing the main data structures we use to represent Wasm's concolic values, stacks, and memories; and **(4)** a dedicated first order solver used to encode the logic of WASP into the logic of its underlying SMT solver, Z3 [24].

Let us now take a look at how WASP concolically executes the Wasm program given as input. Firstly, the given program is parsed by our *Extended Parser*, generating an abstract syntax tree that is then passed to the *Concolic Interpreter*. The Concolic Interpreter implements the main concolic execution loop exploring one execution path at a time and generating for each path its corresponding path condition. The interpreter executes the given program by concolically evaluating one instruction at a time following the small-step concolic semantics presented in §3.2. Concolic execution requires the interpreter to keep track of both the program's concrete and symbolic states. To this end, we combine the concrete domains of the original reference interpreter with new symbolic domains modelling Wasm's symbolic values, stacks and memories. At the end of each concolic iteration, the Concolic Interpreter must interact with Z3 to determine the concrete values of the symbolic inputs for the next concolic iteration. This requires converting the logical formulas constructed by WASP into the logic of Z3. This is done by a dedicated *First Order Solver* that essentially translates WASP formulas into Z3 formulas using the Z3 OCaml bindings.

3.2 Concolic Execution Semantics

We define a concolic semantics of Wasm, which we use to guide the implementation of the concolic interpreter at the core of WASP. Our semantics operates on concolic states, which can be viewed as pairs of concrete and symbolic states. Concolic states are therefore inhabited by both concrete and symbolic values. Formally, symbolic values are given by the grammar:

$$\hat{s} := c \mid \hat{x} \mid \ominus(\hat{s}) \mid \oplus(\hat{s}, \hat{s}) \mid \otimes(\hat{s}, \hat{s}, \hat{s})$$

Symbolic values include: Wasm concrete values c, symbolic variables $\hat{x}$, and various Wasm unary and binary operators, respectively ranged by $\ominus$ and $\oplus$. Additionally, there is a ternary operator $\otimes$ reserved for symbolic byte expressions. As discussed above, we extended the syntax of Wasm with various instructions for creating and reasoning over symbolic values. In the formalism, we model the following three instructions:

$$e ::= \cdots \mid \texttt{sym_assume} \mid \texttt{sym_assert} \mid t.\texttt{symbolic}$$

Where: $t.$`symbolic` is used to create a symbolic value of type t; `sym_assume` is used to add the constraint on top of the stack to the current path condition; and `sym_assert` is used to check whether the constraint on top of the stack is implied by the current path condition.

Before proceeding to the description of the concolic semantics, we must first define concolic states. A concolic state is composed of: **(1)** a *concolic memory* μ, mapping integer addresses to pairs of concrete bytes and symbolic bytes; **(2)** a *concolic local store* ρ, mapping local variable indexes to pairs of concrete and symbolic values (e.g., $\rho = [0 \mapsto (2, \hat{y})]$); **(3)** a *concolic global store* δ mapping global variable indexes to pairs of concrete and symbolic values (e.g., $\delta = [0 \mapsto (2, \hat{y})]$); **(4)** a *concolic stack* st, consisting of a sequence of pairs of concrete and symbolic values (e.g., $st = (2, \hat{y}) :: (0, \hat{x})$); **(5)** a *symbolic environment* ε mapping symbolic variables to concrete values (e.g., $\varepsilon = [\hat{x} \mapsto 0, \hat{y} \mapsto 2]$); and **(6)** a *path condition* π keeping track of all the constraints on which the current execution has branched so far. All concolic domains are obtained by lifting the respective concrete domains, as defined in [30], from concrete values to pairs of concrete and symbolic values. For instance, while a concrete local store maps local variable indexes to concrete values, a concolic local store maps local variable indexes to pairs of concrete and symbolic values. In contrast to the concolic domains, symbolic environments do not have a counterpart in concrete execution. The concolic interpreter uses the symbolic environment to link the program's symbolic variables to their concrete values, essentially storing the bindings of the symbolic variables computed at the beginning of each concolic iteration.

The concolic semantics makes use of computation *outcomes* [26] to capture the flow of execution. We consider five types of outcomes: **(1)** the non-empty continuation outcome `Cont`(e), signifying that the execution of the current instruction generated a new instruction to be executed next; **(2)** the empty continuation outcome `Cont`, signifying that the execution may proceed to the next instruction; **(3)** the trap outcome `Trap`, signifying that the execution of the current instruction generated a Wasm trap; **(4)** the failed assertion outcome `AsrtFail`, signifying that the execution of the current instruction resulted in an assertion failure; and **(5)** the failed assumption outcome `AsmFail`, signifying that the execution of the current instruction resulted in an assumption failure. The concolic domains are summarised below.

Concolic Semantic Domains

Local Store	$\rho : i32 \to c \times \hat{s}$		
Stack	$st : (c \times \hat{s})$ *list*	Outcome	$o ::=$ `Cont`(e) | `Cont` | `Trap` | `AsrtFail` | `AsmFail`
Logical Env	$\varepsilon : \hat{x} \to c$		
Path Cond	π	Symbolic Expr	$\hat{s} ::= c \mid \hat{x} \mid \ominus(\hat{s}) \mid \oplus(\hat{s}, \hat{s}) \mid \otimes(\hat{s}, \hat{s}, \hat{s})$
Global Store	$\delta : i32 \to c \times \hat{s}$		
Memory	$\mu : i32 \to c \times \hat{s}$		

$$\textsc{Load}\quad \frac{n = \mathtt{size}(t) \qquad (c, \hat{s}') = \mathtt{load_bytes}(\mu, k+o, n)}{t.\mathtt{load}\ o, (k, \hat{s}) :: st, \mu \ \Rightarrow_{\mathsf{cs}}\ (c, \hat{s}') :: st, \mu, \mathtt{Cont}}$$

$$\textsc{Store}\quad \frac{\mu' = \mathtt{store_bytes}(\mu, k+o, (c, \hat{s}))}{t.\mathtt{store}\ o, (k, \hat{s}_k) :: (c, \hat{s}) :: st, \mu \ \Rightarrow_{\mathsf{cs}}\ st, \mu', \mathtt{Cont}}$$

$$\textsc{GetLocal}\quad \frac{}{\mathtt{get_local}\ i, \rho, st \ \Rightarrow_{\mathsf{cs}}\ \rho, \rho(i) :: st, \mathtt{Cont}}$$

$$\textsc{SetLocal}\quad \frac{\rho' = \rho[i \mapsto (c, \hat{s})]}{\mathtt{set_local}\ i, \rho, (c, \hat{s}) :: st \ \Rightarrow_{\mathsf{cs}}\ \rho', st, \mathtt{Cont}}$$

$$\textsc{SymAssert-CFail}\quad \frac{c = 0}{\mathtt{sym_assert}, \rho, (c, \hat{s}) :: st, \varepsilon, \pi \ \Rightarrow_{\mathsf{cs}}\ \mathtt{AsrtFail}}$$

$$\textsc{SymAssert-SFail}\quad \frac{c \neq 0 \qquad (\pi \wedge (\hat{s} = 0))\ \mathsf{SAT}}{\mathtt{sym_assert}, (c, \hat{s}) :: st, \pi \ \Rightarrow_{\mathsf{cs}}\ st, \pi, \mathtt{AsrtFail}}$$

$$\textsc{SymAssert-Pass}\quad \frac{c \neq 0 \qquad (\pi \wedge (\hat{s} = 0))\ \mathsf{UNSAT}}{\mathtt{sym_assert}, (c, \hat{s}) :: st, \pi \ \Rightarrow_{\mathsf{cs}}\ st, \pi, \mathtt{Cont}}$$

$$\textsc{SymAssume-Fail}\quad \frac{st = (c, \hat{s}) :: st' \qquad c = 0 \qquad \pi' = \pi \wedge (\hat{s} = 0)}{\mathtt{sym_assume}, st, \pi \ \Rightarrow_{\mathsf{cs}}\ st', \pi', \mathtt{AsmFail}}$$

$$\textsc{SymAssume-Pass}\quad \frac{st = (c, \hat{s}) :: st' \qquad c \neq 0 \qquad \pi' = \pi \wedge (\hat{s} \neq 0)}{\mathtt{sym_assume}, st, \pi \ \Rightarrow_{\mathsf{cs}}\ st', \pi', \mathtt{Cont}}$$

$$\textsc{Symbolic-Fresh}\quad \frac{\hat{x} \notin \mathsf{dom}(\varepsilon) \qquad i \in t \qquad \varepsilon' = \varepsilon[\hat{x} \mapsto i]}{t.\mathtt{symbolic}\ \hat{x}, st, \varepsilon \ \Rightarrow_{\mathsf{cs}}\ (i, \hat{x}) :: st, \varepsilon', \mathtt{Cont}}$$

$$\textsc{Symbolic}\quad \frac{\hat{x} \in \mathsf{dom}(\varepsilon)}{t.\mathtt{symbolic}\ \hat{x}, st, \varepsilon \ \Rightarrow_{\mathsf{cs}}\ (\varepsilon(\hat{x}), \hat{x}) :: st, \varepsilon, \mathtt{Cont}}$$

Figure 4 Fragment of WebAssembly concolic semantics: non-control-flow instructions.

We formalise the concolic semantics of Wasm instructions using a semantic judgement of the form: $e, \rho, st, \varepsilon, \pi, \delta, \mu \ \Rightarrow_{\mathsf{cs}}\ \rho', st', \varepsilon', \pi', \delta', \mu', o$ meaning that the concolic evaluation of the instruction e in the local store ρ, stack st, symbolic environment ε, path condition π, global store δ, and memory μ results in a new local store ρ', stack st', logical environment ε', path condition π', global store δ', memory μ', and outcome o. Figures 4 and 5 present a selection of the semantic rules. In the presentation of the rules, we omit the elements of the configuration that are neither updated nor inspected by the current rule, writing, for instance, $e, \rho, st \ \Rightarrow_{\mathsf{cs}}\ \rho', st', o$ to mean $e, \rho, st, \varepsilon, \pi, \delta, \mu \ \Rightarrow_{\mathsf{cs}}\ \rho', st', \varepsilon, \pi, \delta, \mu, o$. The selected concolic rules are explained below.

Load. This rule first computes the concrete address whose value is to be loaded from memory by adding the given offset parameter o to the concrete memory address k at the top of the stack. Then, the concolic pair stored at the real memory address $k + o$ is loaded from memory using the auxiliary function `load_bytes` and placed at the top of the stack. The function `load_bytes`, explained in §3.3, receives as parameter not only the memory μ and the concrete address $k + o$ but also the size n of the memory chunk to be loaded, which is determined using the auxiliary function `size`.

Store. This rule pops the first two concolic pairs out of the stack, with the second one denoting the value to be stored and the first one the memory address where to store it. Then, the rule computes the real memory address $k + o$ by adding the given offset parameter o to the concrete address k. Next, it uses the function `store_bytes`, explained in §3.3, for storing the concolic pair $(c, \hat{s})$ at $k + o$. In contrast to `load_bytes`, the function `store_btyes` does not require the size of the value to be stored which can be determined from the value c.

SymAssert. The SymAssert rules look at the value c on top of the stack $(c, \hat{s}) :: st$. If $c = 0$ (CFail), it immediately raises `AsrtFail` and the interpreter stops. Otherwise, it checks if that the current path condition implies that the the value on top of the stack is different

$$\frac{st = (c, \hat{s}) :: st' \quad c \neq 0 \quad \pi' = \pi \wedge (\hat{s} \neq 0) \quad o = \texttt{Cont (block } tf\ e_1^*\texttt{)}}{\texttt{if } tf\ e_1^* \texttt{ else } e_2^*, st, \pi \Rightarrow_{\mathsf{cs}} st', \pi', o}\ \textsc{If-True}$$

$$\frac{st = (c, \hat{s}) :: st' \quad c = 0 \quad \pi' = \pi \wedge (\hat{s} = 0) \quad o = \texttt{Cont (block } tf\ e_2^*\texttt{)}}{\texttt{if } tf\ e_1^* \texttt{ else } e_2^*, st, \pi \Rightarrow_{\mathsf{cs}} st', \pi', o}\ \textsc{If-False}$$

$$\frac{st = (k, \hat{s}) :: st' \quad \pi' = \pi \wedge (\hat{s} = k)}{\texttt{br_table } j_1^k\ j\ j_2^*, st, \pi \Rightarrow_{\mathsf{cs}} st', \pi', \texttt{Cont (br } j\texttt{)}}\ \textsc{Tbl-Brk-In}$$

$$\frac{st = (c, \hat{s}) :: st' \quad c \geq k \quad \pi' = \pi \wedge (\hat{s} \geq k)}{\texttt{br_table } j_1^k\ j, st, \pi \Rightarrow_{\mathsf{cs}} st', \pi', \texttt{Cont (br } j\texttt{)}}\ \textsc{Tbl-Brk-Out}$$

$$\frac{st = (j, \hat{s}) :: st' \quad \pi' = \pi \wedge (\hat{s} = j) \quad funcs(j) = \left(\texttt{func } tf \texttt{ local } t^*\ e^*\right)}{\texttt{call_indirect } tf, st, \pi \Rightarrow_{\mathsf{cs}} st', \pi', \texttt{Cont (call } j\texttt{)}}\ \textsc{Call Indirect - Found}$$

$$\frac{st = (j, \hat{s}) :: st' \quad j \notin \mathsf{dom}_{tf}(funcs) \quad \pi' = \pi \wedge (\hat{s} \notin \mathsf{dom}_{tf}(funcs))}{\texttt{call_indirect } tf, st, \pi \Rightarrow_{\mathsf{cs}} st', \pi', \texttt{Trap}}\ \textsc{Call Indirect - Not-Found}$$

Figure 5 Fragment of WebAssembly concolic semantics: control-flow instructions.

from 0; formally: $\pi \Rightarrow (\hat{s} \neq 0)$. Checking the validity of $\pi \Rightarrow (\hat{s} \neq 0)$ is equivalent to checking the satisfiability of $\neg(\pi \Rightarrow (\hat{s} \neq 0))$; formally: $\pi \Rightarrow (\hat{s} \neq 0)$ is valid *if and only if*, $\neg(\pi \Rightarrow (\hat{s} \neq 0))$ is **not** satisfiable. Simplifying $\neg(\pi \Rightarrow (\hat{s} \neq 0))$, we obtain the formula $\pi \wedge (\hat{s} = 0)$. Hence, the rule checks if the formula $\pi \wedge (\hat{s} = 0)$ is satisfiable, in which case (SFail) the assertion fails and the outcome `AsrtFail` produced; otherwise (Pass), the assertion holds and the program may continue, as given by the outcome `Cont`.

SymAssume. The SymAssume rules check the value c on top of the stack, which is expected to be different from 0. Hence, if $c = 0$ (Fail), the current concolic iteration can be discarded as it is not relevant to the programmer. To achieve this, the semantics leaves the current concolic state unchanged, generating the outcome `AsmFail` and extending the current path condition with the formula $\hat{s} = 0$. If $c \neq 0$ (Pass), the concolic execution may proceed, simply conjuncting the formula $\hat{s} \neq 0$ with the current path condition.

Symbolic. The Symbolic-Fresh and Symbolic rules are used for the creation of a symbolic variable of the type t, named $\hat{x}$. If the variable $\hat{x}$ is already present in the mappings of the symbolic environment, $\hat{x} \in \mathsf{dom}(\varepsilon)$, then this variable already exists and its mapped value is inserted on top of the stack $(\varepsilon(\hat{x}), \hat{x}) :: st$. If $\hat{x}$ does not exist in the symbolic environment a new entry is created, where $\hat{x}$ is mapped to a random value i of type t, resulting in the new symbolic environment $\varepsilon' = \varepsilon[\hat{x} \to i]$, and $(i, \hat{x})$ being put on top of the stack.

If. The If rule analyses the concrete value c on top of the stack $(c, \hat{s}) :: st$. If $c \neq 0$, then the path condition is conjoined with the symbolic expression associated with the value on top of the stack $\pi \wedge (\hat{s} \neq 0)$. The resulting outcome is a `block` with the set of instructions e_1^*, corresponding to the "then" branch. If $c = 0$, the opposite happens, the resulting path condition is $\pi \wedge (\hat{s} = 0)$, and the outcome is a block with the set of instructions e_2^*, corresponding to the "else" branch.

Table-Break. The Tbl-Brk rules first check the integer value k at the top of the stack and then inspect the list, $j_1, \ldots, j_n$, of argument indices. If $k \leq n$ (Table-Break-In), the semantics simply obtains the $(k+1)$-th index, j_{k+1}, and returns the outcome `Cont` (`br` j_{k+1}) to transfer the control to the j_{k+1} enclosing block. If $k > n$ (Table-Break-Out), the semantics proceeds as in the previous case but with the index j_n. For instance, the execution of (`br_table` $4, 3, 2, 1$) with the integer 2 on top of the stack generates the outcome `Cont` (`br` 2), while its execution with 7 on top of the stack generates `Cont` (`br` 1). At the symbolic level, both rules extend the path condition with information about index taken. The concolic semantics of `br` follows directly from its concrete semantics given in [30] as this rule does not interact with the symbolic elements of the concolic state.

Algorithm 1 Concolic interpreter main loop.

```
1 function ConcolicExecute(e, ρ, st, δ, μ)
2   Π, π ← true, true
3   while Π is SAT ∧ belowLimit() do
4     ε_i ← Model(π)
5     e, ρ, st, ε_i, π, δ, μ ⇒*_cs ρ′, st′, ε′, π′, δ′, μ′, o
6     if o = Error then
7       return false
8     Π ← Π ∧ ¬π′
9   return true
```

Call-Indirect. The Call Indirect rules first check the integer index j at the top of the stack and inspect the function table to obtain the corresponding function. The Found rule models the case in which the j-th function exists and its type coincides with the one provided to the call instruction. In this case, the semantics simply generates the outcome `Cont` (`call` j), indicating that the j-th function is to be executed next with no extra check. The Not-Found rule models the case in which either the j-th function does not exist or its type does not coincide with the given argument. In this case, the semantics generates the `Trap` outcome. At the symbolic level, both rules extend the current path condition. The Found rule is straightforward, simply recording the index of the executed function. The Not-Found rule is slightly more convoluted. Instead of simply recording the failing index ($\hat{s} = j$), it records all possible failing indexes ($\hat{s} \notin \mathsf{dom}_{tf}(\mathit{funcs})$), preventing the concolic loop from generating concrete inputs that lead to new illegal calls at that execution point. Analogously to the `br` instruction, the concolic semantics of `call` also follows directly from its concrete semantics given in [30].

3.2.1 Concolic Loop

Concolic execution engines execute a given program multiple times in order to explore all possible execution paths. Algorithm 1 presents WASP's main concolic loop. To generate new concrete inputs at the end of each concolic iteration, the concolic interpreter maintains a *global path condition* Π representing all the execution paths that remain to be explored. At the beginning of each concolic iteration, the satisfiability of the global path condition is checked with the help of Z3. If Π is satisfiable, Z3 returns a model, which the engine uses to construct a new symbolic environment, mapping the symbolic variables of the program being analysed to new concrete values. If Π is not satisfiable, the execution stops, given that all possible execution paths have already been explored. Initially, Π is set to *true*, meaning that all paths still have to be explored. At the end of each iteration, the engine updates the global path condition to $\Pi \wedge \neg\pi'$, where π' is the final path condition of the iteration at hand. By adding $\neg\pi'$ to the global path condition, we prevent that future concolic iterations go down the same execution path as the current iteration.

3.2.2 Concolic Execution Example

To illustrate how the proposed concolic semantics works in practice, let us consider the Wasm program given in Figure 6a. This program results from the compilation of the C program given in Figure 2a. For clarity, we represent the given program in Wasm Textual Format (WAT), in which program variables are associated with string identifiers instead of

```
1  (func $main
2    ;; init x=symb(), y=symb(),
3    ;;      a=4, b=2...
4    (i32.const 0)
5    (i32.gt_s)
6    (if
7      (then
8        (i32.const 6)
9        (local.set $b)
10       (local.get $x)
11       (local.get $y)
12       (i32.lt_s)
13       (if
14         (then
15           (local.get $x)
16           (i32.const 2)
17           (i32.mul)
18           (local.get $y)
19           (i32.add)
20           (local.set $a)))))
21   (local.get $a)
22   (local.get $b)
23   (i32.ne)
24   (sym_assert))
```

(a) Wasm program for Listing 2a.

(b) Concolic execution flow diagram for the program in Listing 6a, executing the sequence of instructions indicated in lines: 4-6, and 21-24.

Figure 6 Concolic execution example in WASP.

integer indices. Figure 6b illustrates the concolic iteration corresponding to the right-most path of the execution tree in Figure 2b, with each node in the flow diagram representing a configuration of the concolic semantics. We represent the path condition π, local store ρ, execution stack st, and the current instruction to be executed. The symbolic environment, linear memory, and global store are not represented as they are not manipulated by the program. Specifically, the execution depicted in Figure 6b represents a concolic iteration of the program where the symbolic variables x and y are concretely assigned to 0 and 2, respectively. In this case, the first if-statement of the corresponding C program (i.e., $x > 0$) is false, causing the program to jump to the final assert instruction which holds (i.e., $a \neq b$). In the compiled Wasm program listed in Figure 2a, this if-statement corresponds to the Wasm instructions shown in lines 4-6. The concolic execution of these instructions leads to the first three configurations depicted in the flow diagram (confs. 1-3). The final assertion is translated to the lines 21-24 of the Wasm program, whose execution corresponds to the last four configurations in the flow diagram (confs. 4-7). At the end of the concolic iteration, WASP negates the obtained path condition, $\pi = \hat{x} \leq 0$, in order to generate the inputs for the next concolic iteration as explained in the previous sub-section.

3.3 Symbolic Memory

To concolically execute Wasm code, WASP requires the ability to reason at the byte-level granularity. This requirement is important because Wasm code often needs to operate over the in-memory representation of data at the finer-grained level of bytes or bits. One such example is given in Listing 7b, which shows a real-world function for converting the endianness of a 32-bit unsigned integer. To help explain the example, let us first consider the corresponding C function given in Listing 7a. This example receives an unsigned integer parameter `x` and returns the unsigned integer obtained by swapping the order of the bytes of `x`. Before proceeding to the description of the example, recall that: **(1)** the union data

```c
unsigned int swap(unsigned int x) {
  union {
    unsigned int i; char c[4];
  } src, dst;
  src.i = x;
  dst.c[3] = src.c[0];
  dst.c[2] = src.c[1];
  dst.c[1] = src.c[2];
  dst.c[0] = src.c[3];
  return dst.i;
}
```

(a) Endianness swap function, taken from [46].

```wasm
(func $swap (param $x i32) (result i32)
  (local $src i32) (local $dst i32)
  (local.get $src)
  (local.get $x)
  (i32.store)
  (local.get $dst)
  (local.get $src)
  (i32.load8_u offset=0)
  (i32.store8  offset=3)
  ;; ...
  (return))
```

(b) Snippet of the Wasm program resulting from the compilation of the program in Listing 7a.

Figure 7 Symbolic byte manipulation example.

type is used for storing different data types in the same memory segment; **(2)** in a standard 32-bit architecture, characters are represented by one byte and integers by four bytes; and **(3)** local variables are stored in the stack segment of the C memory.

The `swap` function first declares two variables `src` and `dst`, which can hold either an unsigned integer or an array of four characters. Note that, the two members of this union take exactly the same space, 4 bytes. Then, it copies the four bytes of `x` to the segment of memory referenced by `src`. Next, it copies each individual byte of `src` to the segment of memory referenced by `dst` in reverse order; that is the last byte of `src` will be the first byte of `dst` and so on and so forth. Finally, the function returns the integer value of `dst`.

Note that, the same segment of memory is accessed differently depending on the member of the union type that is used to interact with it. If one uses the union member `i`, one reads/writes four bytes from/into the corresponding memory segment. Conversely, if one uses the union member `c`, one reads/writes a single byte from/into the corresponding memory segment. This example clearly demonstrates the need for byte-level reasoning in SE tools.

Byte-Level Operators. In order to reason about byte-level memory operations, we make use of the operators `concat` and `extract`, which work as follows:

- The expression $\texttt{concat}(\hat{s}_1, \hat{s}_2)$ denotes the bit-vector resulting from the concatenation of the bit-vectors denoted by $\hat{s}_1$ and $\hat{s}_2$. For instance, $\texttt{concat}(\texttt{0xBE}, \texttt{0xEF}) = \texttt{0xBEEF}$.
- The expression $\texttt{extract}(\hat{s}, h, l)$ denotes the bit-vector corresponding to the bytes of the bit-vector denoted by $\hat{s}$ that occur in $[l, h[$. This means that the expression $\texttt{extract}(\hat{s}, h, l)$ denotes a bit-vector of size $h - l$. For instance, $\texttt{extract}(\texttt{0xBEEF}, 1, 0) = \texttt{0xEF}$.

Given that our underlying Z3 encoding represents all primitive types as bit-vectors, the encoding of these operators into the logic of Z3 is trivial as they have equivalent Z3 operators.

Byte-Addressable Memory. Note that our concolic Wasm memory is a mapping from integer indexes, representing concrete memory addresses, to pairs of concrete and symbolic bytes. This means that before we store a given value in memory, we have to obtain the expressions denoting its corresponding bytes. Conversely, when loading a primitive type from memory, we must concatenate the symbolic expressions denoting its component bytes to obtain the symbolic expression that denotes the full value. To do this, we enlist two helpers:

- The function $\texttt{store_bytes}(\mu, l, (c, \hat{s}))$, that individually unpacks each concrete and symbolic byte from the value pair $(c, \hat{s})$, using the `extract` operator, and then sequentially

stores the obtained concrete and symbolic bytes into the segment of μ pointed to by the l, resulting in a new symbolic memory μ'.

- The function `load_bytes`(μ, l, n), that sequentially loads n concrete and symbolic bytes from the concolic memory at address l and concatenates them using the `concat` operator, resulting in a new concolic pair of the form $(c, \hat{s})$.

We mathematically formalise the functions `store_bytes` and `load_bytes` in the table below.

Memory Operations

StoreBytes

$$\frac{n = |c| \qquad c_i = \texttt{extract}(c, i, i-1)|_{i=1}^{n} \qquad \hat{s}_i = \texttt{extract}(\hat{s}, i, i-1)|_{i=1}^{n}}{\texttt{store_bytes}(\mu, l, (c, \hat{s})) = \mu\,[l + (i-1) \mapsto (c_i, \hat{s}_i)]\,|_{i=1}^{n}}$$

LoadBytes

$$\frac{(c_i, \hat{s}_i) = \mu(l + (i-1))|_{i=1}^{n} \qquad c = \texttt{concat}(c_1, \ldots, c_n) \qquad \hat{s} = \texttt{concat}(\hat{s}_1, \ldots, \hat{s}_n)}{\texttt{load_bytes}(\mu, l, n) = (c, \hat{s})}$$

Byte-level Simplifications. While the concrete application of the operators `extract` and `concat` always yields a fully resolved concrete value, it is often not possible to resolve the application of these operators to symbolic values. For instance, the application of `concat` to two symbolic values $\hat{s}_1$ and $\hat{s}_2$ simply yields the symbolic expression `concat`$(\hat{s}_1, \hat{s}_2)$. As every time WASP interacts with the heap, it applies byte-level operators to the values being stored or loaded, concolic execution rapidly increases the complexity of the symbolic expressions handled by the program. This constitutes a serious problem as the additional complexity introduced by byte-level operators is detrimental to the overall performance of WASP. To counter this issue, we apply two simple algebraic simplifications to symbolic values, every time a symbolic value is loaded from memory. The simplification rules are captured by the following algebraic identities:

$$h - l = \texttt{size}(\texttt{type}(\hat{s})) \Rightarrow \texttt{extract}(\hat{s}, h, l) = \hat{s} \tag{1}$$

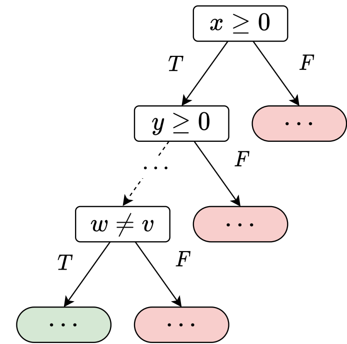
$$\texttt{concat}(\texttt{extract}(\hat{s}, h, m), \texttt{extract}(\hat{s}, m, l)) = \texttt{extract}(\hat{s}, h, l) \tag{2}$$

3.4 Shortcut Restarts

Programmers often need to test their functions not for all inputs but only for those that satisfy a specific set of constraints. In WASP, this can be done using the `sym_assume` instruction, which filters out all execution paths for which the symbolic inputs do not satisfy the given constraints. As explained in §3.2, whenever the symbolic interpreter encounters an assume statement whose constraint does not hold, it discards the current concolic iteration as it is not relevant for the developer. This design is, however, inherently inefficient as it requires WASP to restart the concolic execution of the program every time an assumption fails. To help understand this problem, let us consider the C program given in Figure 8a. This program starts with a sequence of n assumptions over its five symbolic variables. In the worst-case scenario, where every assumption fails, WASP would have to restart the analysis n times before actually starting executing the program. As a result, WASP would have to execute $O(n^2)$ lines of code and query Z3 n times before reaching the first meaningful concolic iteration, as illustrated by the execution tree given in Figure 8b.

```
int function_test() {
    int x = symb(), y = symb();
    int u = symb(), v = symb();
    int w = symb();
    // Setup assumptions (Asm)
    sym_assume(x >= 0);    // Asm₁
    sym_assume(y >= 0);    // Asm₂
    ...
    sym_assume(w != v);    // Asmₙ
    // Program starts here
    ...
}
```

(a) Chained assumptions that do not affect the path condition set.



(b) Execution tree for the example in Listing 8a, where the red nodes imply a restart in WASP.

Figure 8 Example of assumption handling in WASP.

To solve this problem, we propose an adaptation of the concolic semantics of SymAssume given in Figure 4, which avoids the need for restarting the concolic execution from the beginning of the program whenever a failed assumption is reached. The concolic semantics of the assume instruction is captured by the rules given and described below.

Optimised SymAssume Semantic Rules

SymAssume-Fail
$$\frac{c = 0 \qquad (\pi \wedge \hat{s})\ \mathsf{UNSAT}}{\texttt{sym_assume}, ((c, \hat{s}) :: st), \pi \ \Rightarrow_{\mathsf{cs}}\ \rho, (\neg\hat{s} \wedge \pi), \texttt{AsmFail}}$$

SymAssume-Pass1
$$\frac{c \neq 0}{\texttt{sym_assume}, ((c, \hat{s}) :: st), \pi \ \Rightarrow_{\mathsf{cs}}\ st, (\hat{s} \wedge \pi), \texttt{Cont}}$$

SymAssume-Pass2
$$\frac{c = 0 \qquad \varepsilon' = \texttt{model}(\pi \wedge \hat{s}) \qquad (\rho', st', \delta', \mu') = \texttt{update_model}(\varepsilon', (\rho, st, \delta, \mu))}{\texttt{sym_assume}, \rho, ((c, \hat{s}) :: st), \varepsilon, \pi, \delta, \mu \ \Rightarrow_{\mathsf{cs}}\ \rho', st', \varepsilon', (\hat{s} \wedge \pi), \delta', \mu', \texttt{Cont}}$$

SymAssume-Fail. This rule is analogous to its previous version given in Figure 4. The difference is that now the current concolic execution is only terminated if there is no model for the conjunction of the current path condition and the formula being assumed, $\pi \wedge \hat{s}$. In this case, the current execution is incompatible with the assumed formula, meaning that it must be discarded.

SymAssume-Pass1. This rule is identical to its previous version given in Figure 4.

SymAssume-Pass2. This rule is the core of our proposed optimisation. It is applied when the current concrete execution does not satisfy the formula being assumed but the current path condition, π, is compatible with the assumed formula, $\hat{s}$. In this case, WASP queries Z3 for a model for $\pi \wedge \hat{s}$ and uses this model to build a new symbolic environment, ε', that satisfies the assumption. Then, WASP has to update all the concolic domains of the program in order for them to be consistent with the new symbolic environment, ε'. To this end, we make use of a function `update_model` that receives as input a symbolic environment ε' and a concolic state, generating a new concolic state, obtained by updating the concrete values of the input state according to the supplied symbolic environment.

3.5 WASP-C

WASP can also be adopted as a tool for indirectly analysing C programs. This section presents WASP-C, a symbolic execution framework to test C programs using WASP. WASP-C takes as input C programs annotated with assumptions and assertions and outputs a *test suite*. A

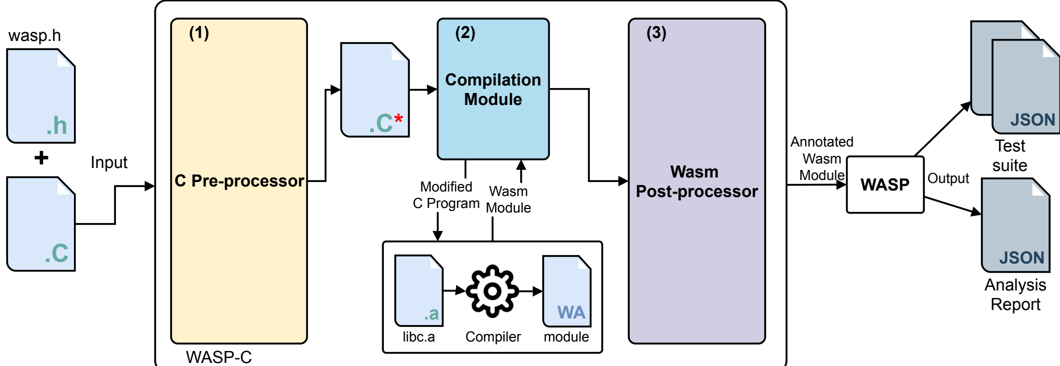


Figure 9 WASP-C high-level architecture.

test suite is a list of test cases, each corresponding to a JSON file, mapping the symbolic variables in the test to their corresponding concrete values. Each test case captures a different execution path of the program to be analysed. Since WASP does not directly operate over the C source code, WASP-C is comprised of three modules whose end goal is to generate a Wasm program for WASP to analyse.

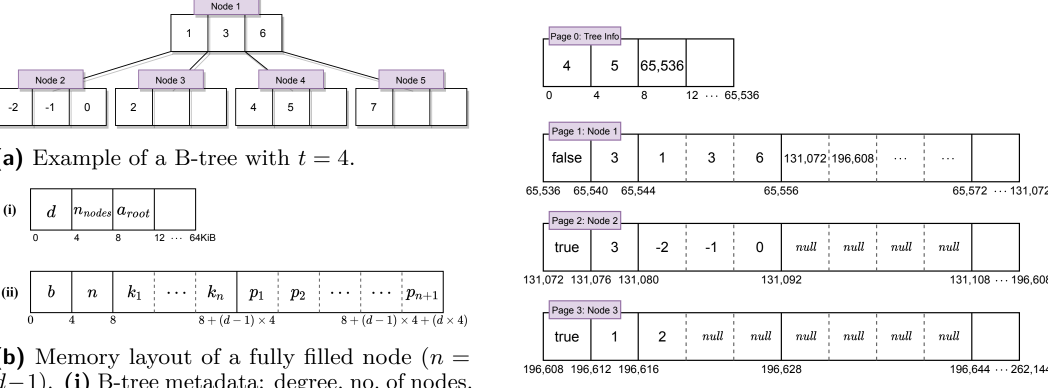
WASP-C is implemented in python and is composed of three essential modules: a *C Pre-processor*, a *Compilation Module*, and a *Wasm Post-processor*, which interact with each other according to the high-level architecture described in Figure 9. Using WASP as a submodule, WASP-C concolically executes C programs as follows. First, the *C Pre-processor* parses the given program using a standard C parser called *pycparser* [8], generating an abstract syntax tree (AST) that is then sent to a specialised C visitor (step 1). Our specialised C visitor traverses the AST, replacing binary operators such as logical ANDs and ORs with specific function calls to avoid spurious branching. Then the AST is exported back to a C program, which is subsequently compiled into Wasm by the *Compilation Module* (step 2). Lastly, the *Wasm Post-processor* processes the obtained Wasm module so as to inject the appropriate WASP symbolic primitives (step 3).

4 Evaluation

We evaluate WASP with respect to five evaluation questions: (EQ1) How does WASP compare to the existing symbolic execution tools for Wasm? (EQ2) Can WASP-C be used to detect bugs in C data structures? (EQ3) Can WASP-C support different types of symbolic reasoning? (EQ4) What are the performance gains of our proposed optimisation techniques? and (EQ5) Can WASP-C scale to industry-grade code?

When it comes to EQ1, we compare WASP against Manticore [51] as it is the only symbolic execution tool that can directly be applied to Wasm binaries; the other existing tool, WANA [73], works only on EOSIO and Ethereum smart contracts, not including a stand-alone symbolic execution engine for Wasm that can be run on its own.

All experiments were performed on a server with a 12-core Intel Xeon E5–2620 CPU and 32GB of RAM running Ubuntu 20.04.2 LTS. For the constraint solver, we employed Z3 v4.8.1. For compiling our benchmarks, we used clang v10.0.0 as part of the LLVM compiler toolchain kit v10.0.0, which includes: *opt*, the LLVM optimiser and analyser; *llc*, the LLVM static compiler; and *wasm-ld*, the Wasm version of *lld*, which is the LLVM object linker. For each execution of WASP, we use the flag `-u` which disables WASP's type checker, set a timeout of 15 minutes, and limit the executing process to 15GiB of memory.



(a) Example of a B-tree with $t = 4$.

(b) Memory layout of a fully filled node ($n = d-1$). **(i)** B-tree metadata: degree, no. of nodes, and root address. **(ii)** B-tree node: leaf node b, no. of keys, key values, and children addresses.

(c) Memory layout of the first three nodes in Figure 10a.

Figure 10 Memory layout of a simple B-tree.

4.1 EQ1: Comparison with Manticore

To compare WASP with Manticore, we use both these tools to symbolically analyse a custom-made Wasm implementation of a B-tree data structure developed by C. Costa [22] and inspired by that of Watt et al. [75]. This data structure allows us to effectively test the scalability of both engines with respect to code size ($\approx$ 4000 LoC) and the complexity of the generated formulas. In the following, we first give a high-level description of the B-tree implementation and the experimental set-up and then present the obtained results.

B-Tree Implementation. B-trees are n-ary self-balancing trees typically used in the implementation of storage systems [55]. B-trees have a fixed branching factor d, denoting the maximum number of children that internal nodes may have. B-tree nodes store at most $d-1$ keys; internal nodes additionally store the pointers to their respective children. Intuitively, it is as if each internal node stores one key in between each two consecutive child pointers. The keys stored inside a B-tree are arranged so that: **(1)** the keys of every node are ordered; and **(2)** each key k_i stored in between the pointers p_i and p_{i+1} of an internal node is greater than all the keys stored in the node pointed to by p_i and less than those stored in the one pointed to by p_{i+1}. B-trees must additionally satisfy various other invariants; the reader is referred to [21] for a thorough account of B-trees and their properties. Figure 10a shows a B-tree with branching factor $d = 4$. The tree contains one internal node (Node 1) and four leaf nodes (Nodes 2 to 5), with the internal node storing three keys. Observe that, for instance, the second key stored in Node 1 (key 3) is greater than the single key stored in Node 3 (key 2) and less than both keys stored in Node 4 (keys 4 and 5).

The B-tree implementation we use [22], as that of Watt et al. [75], only holds 32-bit integer keys. Each B-tree node is kept in a separate memory page according to the memory layout given in Figure 10b. Each memory page stores: **(1)** a flag b, indicating if it represents a leaf node; **(2)** an integer n, denoting the number of keys that the node holds; and **(3)** n keys, $k_1, \ldots, k_n$. Additionally, each internal node stores $n+1$ child pointers, $p_1, \ldots, p_{n+1}$. The implementation uses an extra memory page for keeping metadata information about the B-tree, namely: its branching factor d, number of nodes n_{nodes}, and address of the root node a_{root}. Figure 10c shows the memory layout of the first three nodes of the B-tree presented in Figure 10a together with the extra memory page used to store its meta-information. The B-tree implementation comes with four main functions: **(1)** `$createBTree`(d) for creating an empty B-tree with the specified degree; **(2)** `$insertBTree`(t, k) for inserting the key k into the tree t; **(3)** `$searchBTree`(t, k) for checking if the tree t holds the key k; and **(4)** `$deleteBTree`(t, k) for deleting the key k from the tree t.

Table 1 Results WASP and Manticore applied to our B-tree benchmarks.

	$n_u = 1$				$n_u = 2$				$n_u = 3$			
n_o	n_{paths}	T_{WASP} (s)	T_{Mcore} (s)	$\times T_{WASP}$	n_{paths}	T_{WASP} (s)	T_{Mcore} (s)	$\times T_{WASP}$	n_{paths}	T_{WASP} (s)	T_{Mcore} (s)	$\times T_{WASP}$
2	3	0.14	2.69	×19.2	12	1.19	22.78	×19.1	60	15.54	260.95	×16.8
3	4	0.55	6.31	×11.5	20	5.04	77.13	×15.3	120	47.52	802.42	×16.9
4	5	2.10	11.29	×5.4	30	8.79	170.04	×19.3	210	137.14	1,886.55	×13.8
5	6	1.45	18.95	×13.1	42	16.41	340.32	×20.7	336	286.15	4,041.37	×14.1
6	7	2.40	35.65	×14.8	56	29.05	627.98	×21.6	504	696.35	8,046.52	×11.6
7	8	7.11	54.61	×7.7	72	51.09	1,161.62	×22.7	720	2,003.00	15,803.34	×7.9
8	9	6.90	90.63	×13.1	91	74.53	1,948.36	×26.1	–	–	–	–
9	10	11.18	133.68	×12.0	110	113.74	2,976.56	×26.2	–	–	–	–

Experimental Setup. In order to compare the performance of WASP against that of Manticore, we use the symbolic test suite of [22]. All symbolic tests follow the same code template but use a varying number of symbolic values, of which some are constrained to be ordered. In the following, we use n_o and n_u to denote respectively the number of ordered and unordered symbolic values used in each test.

Results. Table 1 presents the results obtained when running our symbolic test suite with WASP and Manticore. The number of ordered symbolic values used by the tests varies between 2 and 9 and the number of unordered values between 1 and 3; i.e., $2 \leq n_o \leq 9$ and $1 \leq n_u \leq 3$. For each pair (n_o, n_u), we provide: the number of explored paths (n_{paths}); the execution time for WASP (T_{WASP}); the execution time for Manticore (T_{Mcore}); and the WASP speed-up with respect to Manticore. As expected, the number of explored paths increases exponentially with the number of unordered symbolic values. This is reflected in the time taken by both engines to complete the analysis. For instance, WASP takes more than 30 minutes to run the test with $n_o = 7$ and $n_u = 3$, while taking less than one minute to run the test with $n_o = 7$ and $n_u = 2$.

Most significantly, from Table 1, we observe that WASP is consistently faster than Manticore, achieving a speed-up that ranges from 5.4× to 26.2× and averages 15.8×. We conjecture that WASP is able to outperform Manticore for two main reasons: **(1)** Manticore performs static symbolic execution, which means that it interacts more often with the underlying SMT solver and makes more intensive use of memory; and **(2)** Manticore is primarily written in Python which is significantly slower than OCaml.[1]

4.2 EQ2: Detecting Bugs in C Data Structures

To investigate whether WASP-C can detect bugs in complex C data structures, we used it to symbolically test *Collections-C*, a generic data structure library obtained from GitHub [4], which includes a variety of data structures, such as arrays, lists, ring buffers, and queues. In total, it implements ten different data structures spanning just over 11k LoC. The symbolic test suite we used to evaluate WASP on Collections-C comes from the Gillian project [26], in the context of which Collections-C was symbollically tested using Gillian-C, an instantiation of the Gillian framework for the C language. Gillian's authors developed a symbolic test suite that they run against Collections-C. This symbolic test suite consists of 161 symbolic test programs targeting the various data structure algorithms included in Collections-C.

Here, we test two different versions of Collections-C, a version with bugs previously found by the authors of Gillian-C,[2] henceforth *buggy version*, and the version resulting from the

---

[1] `http://roscidus.com/blog/blog/2014/06/06/python-to-ocaml-retrospective`

[2] Version with the 2 bugs identified by Gillian-C: `https://github.com/srdja/Collections-C/pull/119` and `https://github.com/srdja/Collections-C/pull/123`.

Table 2 Results for Gillian-C and WASP-C applied to corrected version of Collections-C.

		Baseline	WASP-C				
Category	n_i	T_{Gil} (s)	T_{WASP} (s)	T_{loop} (s)	T_{solver} (s)	*avg_paths*	$S\left(\frac{T_{Gil}}{T_{WASP}}\right)$
Slist	37	8.34	9.06	6.21	0.85	2	0.92
Pqueue	2	4.79	0.34	0.19	0.05	1	14.09
Stack	2	1.55	0.21	0.06	0.00	1	7.38
Deque	34	8.08	6.43	3.89	1.03	2	1.25
Array	21	7.00	7.00	5.41	1.44	5	1.00
Queue	4	2.11	1.99	1.69	0.18	4	1.06
RingBuffer	3	1.43	0.31	0.07	0.00	1	4.62
Treeset	6	7.07	4.89	4.43	1.43	7	1.45
Treetable	13	12.07	5.02	4.04	1.61	5	2.40
List	37	21.77	30.01	27.18	11.65	6	0.73
Total	159	74.21	65.26	53.17	18.24	34	1.14

correction of those two bugs,[3] henceforth *corrected version*. Essentially, we use WASP-C to execute 161 symbolic test programs developed in the context of the evaluation of the Gillian-C project both against the buggy and corrected version of Collections-C.

Experimental Procedure. We performed two experiments: in the first, we use Gillian-C and WASP-C to execute the symbolic test suite on the corrected version of Collections-C; and in the second, we used the two tools to execute the two error-triggering symbolic tests on the buggy version of Collections-C.

Experiment 1. Table 2 presents the results of Experiment 1, where we use both Gillian-C and WASP-C to test the corrected version of Collections-C. We present the obtained results for each data structure included in Collections-C, showing for each of them: the number of tests (n_i), the total execution time for Gillian-C (T_{Gil}), the total execution time for WASP (T_{WASP}[4]), the total time spent in the concolic interpreter (T_{loop}), the total time in the constraint solver (T_{solver}), the average number of paths explored (*avg_paths*), and the speedup between T_{Gil} and T_{WASP} (S). From the table we observe that, overall, WASP is 1.14× faster than Gillian-C at analysing the complete benchmark suite. And, in 7 out the 10 categories, WASP completes the program analysis faster than Gillian-C (i.e., $T_{WASP} < T_{Gil}$). We conjecture that this performance gain is due to WASP's analysis, i.e., Gillian-C performs static symbolic execution while WASP performs concolic execution, which is faster since it requires fewer interactions with the underlying solver.

During Experiment 1 WASP-C found a new heap-overflow bug in the Pqueue data structure. We confirmed the bug with a concrete test using *AddressSanitizer* [66], reported it to the developers, and fixed it via a pull request, which has already been accepted by the library's main developer.[5] The bug was caused by an integer overflow that subsequently leads to an array-out-of-bounds heap access. WASP-C is able to detect this bug because it models C integers using Z3 bit-vectors, whereas Gillian only used mathematical reals at the time of testing.[6]

Experiment 2. Table 3 presents the results of experiment two, where we use Gillian-C and WASP-C to test the two bug-triggering tests for the buggy version of Collections-C. Since

[3] Corrected version: `https://github.com/srdja/Collections-C`.
[4] Note that, $T_{WASP} = T_{loop} + T_{parse}$ and $T_{loop} = T_{solver} + T_{interpretation}$. Where the parsing and interpretation times, respectively T_{parse} and $T_{interpretation}$, were omitted from the table due to space

Table 3 Bug-finding statistics for Collections-C bugs by WASP and Gillian-C.

Test	Vulnerability	T_{Gil} (s)	T_{WASP} (s)	T_{loop} (s)	T_{solver} (s)	n_{paths}	S
`array_test_remove`	Found	1.40	0.20	0.08	0.03	1	7.00
`list_test_zipIterAdd`	Found	0.57	0.40	0.18	0.00	1	1.42
Total	2/2	1.97	0.60	0.26	0.03	2	3.28

there were only two tests, each triggering a different bug, each row in the table represents a different bug. For each bug, we indicate whether or not WASP found the bug; the remaining columns have the same meaning as in Table 2. As the table indicates, WASP-C is able to detect the bugs discovered by Gillian-C.

4.3 EQ3: Different Types of Symbolic Reasoning

To investigate our third evaluation question, we test WASP-C against the *Test-Comp benchmark suite* (Test-Comp) [10] and compare its results with those obtained for the testing tools submitted to the 2021 Test-Comp Competition [11]. The Test-Comp test suite is organised into different categories with each focusing on a different type of symbolic reasoning. For instance, the categories *Arrays*, *BitVectors* and *Loops* respectively aim at reasoning about array operations, bit-operations, and loops and recursion.

Test-Comp defines two types of testing tasks: **(1)** *Cover-Branches* tasks, whose goal is to generate a set of concrete tests that cover the greatest possible number of program branches, and **(2)** *Cover-Error* tasks, whose goal is to generate at least one set of inputs that lead the execution of the given program to an execution error.

Test-Comp defines a scoring system to classify testing tools depending on how well they perform on both types of tasks. Essentially, a tool is assigned three scores, one for each type of task and a global score. Below, we provide further details on the scoring system.

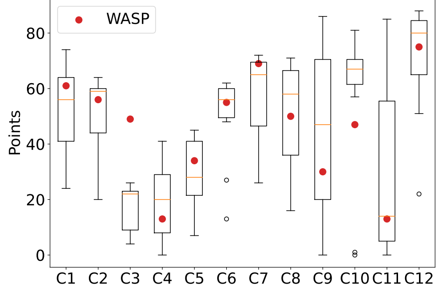
Experimental Setup. We separately evaluate WASP-C on the Cover-Branches and Cover-Error tasks. For each task, we assign WASP-C a global score computed as in Test-Comp. For Cover-Branches, the assigned score represents the coverage of the generated test suites. For Cover-Error, the assigned score represents the number of bugs found. For both tasks we present the results for each testing category (e.g., Arrays, BitVectors, ControlFlow, and so on). We obtain the global score for all analysed categories by applying a weighted average on the individual scores of each category according to the number of tests in that category.

Results. Table 4 presents the evaluation results per category for the Cover-Branches and Cover-Error tasks, respectively, and compares the results obtained for WASP-C with those obtained for the 11 tools submitted to Test-Comp 2021: FuSeBMC [2], CMA-ES Fuzz [40], CoVeriTest [12], HybridTiger [58], KLEE [14], Legion [48], LibKluzzer [44], PRTest [45], Symbiotic [18], TracerX [37], and VeriFuzz [7]. The table shows the minimum and maximum recorded scores obtained for the 11 submitted tools, and the score and rank obtained by WASP-C.[7] Furthermore, in order to better compare WASP-C's results with those of the

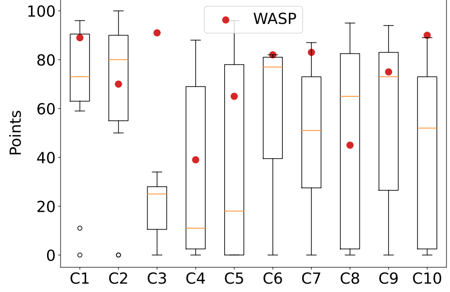
constraints.
[5] Bug fix for heap-overflow bug: `https://github.com/srdja/Collections-C/pull/148`.
[6] Gillian has been since extended with support for mathematical integers and can now detect the bug.
[7] Cover-Error has no results for C11 and C12 because these categories have no Cover-Error tasks.

Table 4 Results for both meta categories: Coverage-Branches and Cover-Error.

Category	Cover-Branches Min	Max	WASP-C	Rank	Cover-Error Min	Max	WASP-C	Rank
C1.Arrays	96	296	245/380	4th	0	96	89/100	4th
C2.BitVectors	13	40	35/57	7th	0	10	7/10	2th
C3.ControlFlow	3	18	33/54	1st	0	11	29/32	1st
C4.ECA	0	12	4/27	8th	0	16	7/18	6th
C5.Floats	16	103	78/202	7th	0	32	21/32	5th
C6.Heap	19	90	80/136	7th	0	47	41/55	7th
C7.Loops	152	424	403/572	4th	0	138	127/156	4th
C8.Recursive	9	38	27/51	8th	0	19	9/20	7th
C9.Sequentialized	0	71	25/39	9th	0	101	75/107	9th
C10.XCSP	0	97	56/100	10th	0	53	54/59	1st
C11.Combinations	0	180	28/210	7th	–	–	–	–
C12.MainHeap	51	204	175/226	8th	–	–	–	–
Score	411	1389	1090	6th	0	405	360	3rd



(a) Box plot for Cover-Branches.



(b) Box plot for Cover-Error.

Figure 11 Box plots: Test-Comp coverage results.

other tools, Figures 11a and 11b plot the results, normalising the scores of all tools in each category and highlighting WASP-C's score with a red dot. For Cover-Branches and Cover-Error, WASP-C ranked sixth and third, respectively, ranking fourth overall. These results demonstrate that WASP-C's symbolic reasoning is on par with state-of-the-art symbolic execution and testing tools for C.

Finally, comparing the CPU time for the top 6 scoring tools in [11], WASP-C was the fifth-fastest tool in the Cover-Error category among these tools (with a total time of 26 hours) and the second-fastest tool in the Cover-Branches category (with a total time of 310 hours). Overall, WASP-C is the second-fastest tool among the top 6 scoring tools, finishing its analyses in about 326 hours and ranking fourth in terms of scoring (note that KLEE, the fastest tool, ranked sixth in terms of scoring). We further note that the tools we compare WASP-C against were executed on a superior testbed. Test-Comp's testbed is a computing cluster consisting of 168 machines; each test-generation run was executed on an otherwise wholly unloaded, dedicated machine to achieve precise measurements. Each machine had one Intel Xeon E3-1230 v5 CPU, with eight processing units each, a frequency of 3.4GHz and 33GB of RAM. This setting is superior to the one in which we run WASP-C: a single server with an Intel Xeon E5–2620 CPU, a frequency of 2.5GHz and 32GB of RAM.

Table 5 Experimental evaluation of OPTMem and OPTRestart.

Benchmark	T_{WASP} (h) on	off	T_{solver} (h) on	off	n_{paths} on	off	n_{solver} on	off	avg_{cmds}(M) on	off	no. ass. fls on	off
OPTMem	32	71	1.80	0.04	16,196	261	15,975	26	603	7.6	106	3
OPTRestart	46	58	17.00	29.00	17,677	73,972	39,810	73,641	132	132	275	255

4.4 EQ4: WASP Optimisations

We introduce two optimisation techniques: **(1)** application of algebraic simplifications to byte-level symbolic expressions generated by memory interactions (§3.3); and **(2)** shortcut restarts for failed assumption statements (§3.4). In the following, we refer to the former technique as OPTMem and to the latter as OPTRestart. To investigate the effectiveness of these optimisations, we compare the execution times obtained for WASP-C when they are enabled (on) against those obtained when they are disabled (off).

Experimental Setup. For each optimisation technique, we have selected a subset of the 2021 Test-Comp benchmark suite [11] with the relevant features. For OPTMem, we have selected the four Test-Comp sub-categories with the highest number of calls to heap-manipulating functions (e.g. `malloc` and `calloc`), whereas for the OPTRestart, we have have selected the four sub-categories with the highest number of assume statements.

Results. Table 5 presents the results obtained for both optimisation techniques, showing: the total execution time of WASP (T_{WASP}); the total solver time (T_{solver}); the total number of explored paths (n_{paths}); the total number of calls to Z3 (n_{solver}); the average number of executed Wasm instructions (avg_{cmd}); and the total number of triggered assertion violations. The values of all metrics are given with the corresponding technique turned on and off.

With OPTMem on, WASP completes the analysis in less 39 hours (32 vs 71), explores 62 times more paths, and discovers 35 times more assertion violations. The main effect of this optimisation is to reduce the size of concolic states by reducing the size of the symbolic expressions that they store. This reduction enables WASP to interpret more instructions per unit of time ($5,234.6$ i/s vs 30.0 i/s), as it has been demonstrated that symbolic execution throughput is severely impacted by intensive memory usage [17].

With OPTRestart on, WASP completes the analyses in less 12 hours and discovers 20 more assertion violations. The main effect of this optimisation is that it allows WASP to explore fewer execution paths by ignoring the paths that lead to failed assumptions, thereby leaving more time for the exploration of relevant paths.

4.5 EQ5: Scalability to Industry-Grade Code

To investigate our fifth evaluation question, we use WASP-C to obtain a comprehensive test suite for part of the C implementation of the *AWS Amazon Encryption SDK* [67], a highly-used cryptographic library that powers, for instance, the Amazon DynamoDB Encryption Client [3]. The AWS Encryption SDK for C is a library for the encryption and decryption of data that implements complex data structures and algorithms in the C language. This library is challenging to analyse as it uses various cryptographic functions that current SMT solvers cannot tackle. The library comes with a benchmark suite of bounded verification proofs designed to be checked with the *CBMC bounded model checker* [42], which can be

Table 6 Benchmark results applying WASP-C to the AWS Encryption SDK for C.

Category	n_i	n_{paths}	T_{loop} (s)	T_{solver} (s)	T_{WASP} (s)	Coverage
Md	2	3	0.12	0.04	0.18	60.8
Decrypt	3	151	48.81	9.03	49.00	54.4
Edk	5	194	366.83	4.64	367.13	60.2
Cmm	5	558	1,793.00	60.43	1,794.00	66.6
Private	3	962	1,792.53	406.06	1,793.11	55.0
Keyring	10	1,382	1,145.89	213.67	1,146.56	70.8
Misc-ops	7	3,851	1,907.59	134.81	1,908.18	48.5
Total	35	7,101	7,054.77	828.68	7,058.16	59.5

easily turned into symbolic tests to enable the generation of a concrete test suite for the library. We consider 35 verification proofs totalling 2.3k LoC. The library itself contains multiple C files totalling just under 40k LoC.

Experimental Procedure. Our experimental procedures analyse the benchmark suite without constraining the number of paths explored during concolic execution and with a timeout of 15 minutes. We choose these settings to enable WASP to freely analyse every path of a program. Note that not all symbolic tests will take 15 minutes to be executed, as some tests do not loop on symbolic values and therefore have a finite execution tree.

Results. Table 6 presents the results of running the created symbolic test suite on seven modules of the AWS Encryption SDK for C organised by the data structure or algorithm that they are testing. Additionally, tests for generic data structures like lists or hash tables and generic operations like getters/setters go into the Misc-ops module as they are not specific to the encryption library. For each module, we present the number of tests targeting that module (n_i), the total number of paths explored (n_{paths}), the total time spent in the concolic loop (T_{loop}), the total time spent in Z3 (T_{solver}), the total analysis time (T_{WASP}), and the line coverage obtained. The table shows that, in total, WASP-C analyses the benchmark suite in just under two hours and obtains roughly 60% of line coverage of the library's functions. In contrast to the data structures in *Md*, which are mainly populated with concrete values and are therefore quickly analysed, the data structures *Edk*, *Cmm*, *Private*, and *Keyring* take a significant amount of time to be analysed as they mainly operate on symbolic values. Unsurprisingly, the symbolic tests in the former group trigger much fewer symbolic execution paths than those in the latter. Note that analyses finish quickly in the Decrypt module that tests decryption operations due to the small inputs given to these operations, typically, strings with one or two characters at most.

As our symbolic test suite is automatically obtained from the bounded verification proofs that come with the AWS Encryption SDK for C, its coverage is limited by the structure of the bounded inputs considered in the proofs. As most proofs only consider well-formed inputs, our symbolic test suite does not cover most of the library's code for handling ill-formed inputs. In the future, we plan to write additional tests to obtain 100% line coverage.

5 Related Work

Semantics of Wasm. Haas et al. [30] proposed a small-step operational semantics for Wasm together with a type system for checking the safety of stack operations. Later, Watt [74] mechanised both the semantics and the type system introduced in [30] using the Isabelle

theorem prover [53] and exposing several issues in the official Wasm specification. The authors of [75] then introduced Wasm Logic, a program logic for modular reasoning about heap-manipulating Wasm programs. In contrast to Wasm's native type system, Wasm Logic can be used to establish the safety of heap operations. However, it cannot yet reason about real-world Wasm code as it has not been automated. Very recently, Watt et al. [76] introduced two new mechanisations of the specification of Wasm following the new official W3C standard [72]; one developed in Isabelle and the other in the Coq [70] theorem prover.

Program Analyses for Wasm. Since the proposal of the Wasm standard [57], various program analyses have been designed for tackling the specificities of the language. Most of these analyses aim at the verification/testing of security properties and can broadly be divided into two main categories: *static analyses* [77, 68], which analyse stand-alone Wasm modules without the need to execute them, and *dynamic analyses* [69, 68], which instrument the given module to enforce the desired security property. Among the static analyses, we highlight:

- CT-Wasm [77], a type-driven extension of Wasm for provably secure implementation of cryptographic algorithms, which enforces information flow security and resistance to timing side-channel attacks through the use of security types;
- Wassail [68], an information flow analysis for Wasm based on a standard data-flow analysis, which the authors evaluate on a benchmark comprising 30 C programs.

Neither CT-Wasm nor Wassail can precisely reason about Wasm programs that interact with the memory, as they both assume that the values stored in memory are always confidential. In the future, we would like to study how to take advantage of WASP to improve the precision of information flow analysis for Wasm using, for instance, the self-composition technique [6] for the generation of vulnerability-triggering inputs.

Among the dynamic analyses for Wasm, we highlight the following two taint-tracking tools: TaintAssembly [27] and the tool presented in [69]. TaintAssembly [27] is a modification of the V8 JavaScript engine for performing basic taint tracking by adding a taint label to function parameters, local variables, and linear memory cells. In [69], the authors present a JavaScript virtual machine (VM) to interpret and run Wasm code, capable of monitoring the flow of sensitive information through taint tracking. However, neither TaintAssembly nor the JavaScript VM described in [69] can accurately track information flows in Wasm, as the former does not propagate indirect taint in comparison operators and the latter does not support floating-points in Wasm. Additionally, these tools are not ideal for testing generic Wasm code as they require concrete inputs to trigger the illegal information flows in the given program. A possible direction for future work is to combine WASP with a taint tracking tool, using WASP to generate inputs.

Symbolic Execution. Symbolic execution has been extensively used to find crucial errors and vulnerabilities in a broad spectrum of programming languages, such as C [28], C++ [14], Java [64], and Python [19]. Regarding the Web, there are several state-of-the-art tools for symbolically executing JavaScript code [60, 61, 62, 47, 65], demonstrating the need for such tools for the validation and testing of modern Web applications.

Symbolic execution tools can be divided into two main classes: *static* and *dynamic/concolic* [5]. Static symbolic execution engines, such as [41, 54, 39, 71, 60, 61], explore the entire symbolic execution tree up to a pre-established depth, while concolic execution engines, such as [14, 28, 64, 17, 62, 47, 65], usually work by pairing up a concrete execution with a symbolic execution and exploring one execution path at a time. An advantage of concolic execution over static symbolic execution is that concolic execution requires less frequent interactions

with the solver and a simpler memory model. There is a vast body of research on both static and concolic symbolic execution tools for a wide variety of programming languages, see [5, 15, 16] for comprehensive surveys on the topic. In the following, we give a detailed account of the only two existing symbolic execution tools for Wasm other than WASP.

WANA [73] is a cross-platform smart contract vulnerability detection tool employed to find vulnerabilities in EOSIO [43] and Ethereum smart contracts [38]. WANA is based on static symbolic execution and operates over Wasm bytecode. To detect vulnerabilities in smart contracts, WANA comes with three heuristics for EOSIO smart contracts and four for Ethereum smart contracts. Unlike WASP, WANA lacks a stand-alone symbolic execution engine for Wasm. Hence, it is not possible to run WANA on arbitrary Wasm code without refactoring its internal architecture. For this reason, we were unable to evaluate WANA on our B-tree implementation and compare its performance with those of WASP and Manticore.

Manticore [51] is a symbolic execution framework for binaries and smart contracts. Manticore is highly flexible, supporting a wide range of binaries and computing environments, including Wasm bytecode. When it comes to Wasm, Manticore does not expose dedicated primitives for constructing and reasoning over symbolic values at the source language level. Symbolic inputs and constraints are created as part of a complex Python script that must be written for each test [33], which initialises the symbolic state and calls the appropriate Wasm module. This process does not scale for a broad evaluation, as one would have to manually write a python script for each symbolic test. Nonetheless, we compare the performance of WASP with that of Manticore [51] in the analysis of a stand-alone Wasm implementation of a B-tree data structure, demonstrating that WASP is consistently faster.

6 Conclusion

In this paper, we presented WASP, a novel concolic execution engine for testing Wasm modules. To the best of our knowledge, WASP is the first symbolic execution tool to analyse complex WebAssembly code for general-purpose applications. Prior existing tools for symbolically executing Wasm code [51, 73] were only evaluated on smart contracts, which are simpler and therefore easier to analyse than general-purpose applications. On top of WASP, we also developed WASP-C, a symbolic execution framework to test C programs symbolically using WASP. We have extensively evaluated our tools. Our results show that WASP: **(1)** can detect bugs in complex data structure libraries, being able to find a previously unknown bug in a widely-used generic data structure library for C; **(2)** supports different types of symbolic reasoning, having comparable performance to well-established symbolic execution and testing tools for C; and **(3)** can scale to industry-grade code, being able to generate a high-coverage test suite for the Amazon Encryption SDK for C.

References

1. Syrus Akbary and Ivan Enderlin. Wasmer: Run any code on any client [online]. Accessed 27th-October-2021. URL: `https://wasmer.io`.
2. Kaled M. Alshmrany, Rafael S. Menezes, Mikhail R. Gadelha, and Lucas C. Cordeiro. FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs (Competition Contribution). In *Fundamental Approaches to Software Engineering*, 2021.
3. AWS. Amazon DynamoDB Encryption Client [online]. Accessed 28th-October-2021. URL: `https://docs.aws.amazon.com/crypto/latest/userguide/awscryp-service-ddb-client.html`.

4 Sacha Ayoun, Alexis Marinoiu, and Petar Maksimović. Collections-C for symbolic testing with Gillian-C [online]. Accessed 15th-December-2020. URL: `https://github.com/GillianPlatform/collections-c-for-gillian` [cited 15th December 2020].

5 Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 2018.

6 Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 2011.

7 Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and Venkatesh R. VeriFuzz: Program Aware Fuzzing. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

8 Eli Bendersky. pycparser [online]. Accessed 1st-November-2021. URL: `https://github.com/eliben/pycparser`.

9 John Bergbom. Memory safety: old vulnerabilities become new with WebAssembly. Technical report, Forcepoint, December 2018.

10 Dirk Beyer. International Competition on Software Testing (Test-Comp). In *Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

11 Dirk Beyer. Status Report on Software Testing: Test-Comp 2021. In *Fundamental Approaches to Software Engineering*, 2021.

12 Dirk Beyer and Marie-Christine Jakobs. CoVeriTest: Cooperative Verifier-Based Testing. In *Fundamental Approaches to Software Engineering*, 2019.

13 Ruben Bridgewate. Node v12.3.0 [online]. Accessed 27th-October-2021. URL: `https://nodejs.org/en/blog/release/v12.3.0`.

14 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Conference on Operating Systems Design and Implementation*, 2008.

15 Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *International Conference on Software Engineering*, 2011.

16 Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 2013.

17 Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy*, 2012.

18 Marek Chalupa, Jakub Novák, and Jan Strejcek. Symbiotic 8: Parallel and targeted test generation. *Fundamental Approaches to Software Engineering*, 2021.

19 Ting Chen, Xiao-song Zhang, Rui-dong Chen, Bo Yang, and Yang Bai. Conpy: Concolic execution engine for python applications. In *Algorithms and Architectures for Parallel Processing*, 2014.

20 Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web [online]. Accessed 27th-October-2021. URL: `https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface`.

21 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

22 C. Costa. Concolic execution for WebAssembly. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2020. Master's Thesis.

23 Michael Pradel Daniel Lehmann, Johannes Kinder. Everything Old is New Again: Binary Security of WebAssembly. In *USENIX Security Symposium*, 2020.

24 Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

25 Jonathan Foote. Hijacking the control flow of a WebAssembly program [online]. Accessed 27th-October-2021. URL: `https://www.fastly.com/blog/hijacking-control-flow-webassembly` [cited 27th October 2021].

26 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

27 William Fu, Raymond Lin, and Daniel Inge. Taintassembly: Taint-based information flow control tracking for WebAssembly. *arXiv preprint*, 2018.

28 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

29 GoogleSecurityResearch. Google Chrome 73.0.3683.103 - 'WasmMemoryObject::Grow' Use-After-Free [online]. Accessed 27th-October-2021. URL: `https://www.exploit-db.com/exploits/46968` [cited 27th October 2021].

30 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

31 Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

32 Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 1997.

33 Eric Hennenfent. Symbolically Executing WebAssembly in Manticore [online]. Accessed: 30th-November-2021. URL: `https://blog.trailofbits.com/2020/01/31/symbolically-executing-webassembly-in-manticore`.

34 Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime [online]. Accessed 27th-October-2021. URL: `https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime`.

35 Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference*, 2021.

36 IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019.

37 Joxan Jaffar, Rasool Maghareh, Sangharatna Godboley, and Xuan-Linh Ha. Tracerx: Dynamic symbolic execution with interpolation (competition contribution). *Fundamental Approaches to Software Engineering*, 2020.

38 Bhaskar Kashyap. Introduction to smart contracts [online]. Accessed: 30th-November-2021. URL: `https://ethereum.org/en/developers/docs/smart-contracts`.

39 Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

40 H. Kim. Fuzzing with stochastic optimization. Master's thesis, LMU Munich, 2020. Bachelor's Thesis.

41 James C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

42 Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

43 Daniel Larimer and Brendan Blumer. EOS.IO Technical White Paper [online]. Accessed: 30th-November-2021. URL: `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`.

44 Hoang M. Le. LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution). In *FASE*, 2020.

45 Thomas Lemberger. Plain random test generation with PRTest. *International Journal on Software Tools for Technology Transfer*, 2020.

46 Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012. URL: `https://hal.inria.fr/hal-00703441`.

47 Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

48 Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. Legion: Best-First Concolic Testing (Competition Contribution). In *Fundamental Approaches to Software Engineering*, 2020.

49 Aishwarya Lonkar and Siddhesh Chandrayan. The dark side of WebAssembly [online]. Accessed 27th-October-2021. URL: `https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly` [cited 27th October 2021].

50 Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security Chasms of WASM. Technical report, NCC Group, August 2018.

51 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, 2019. `arXiv:1907.03890`.

52 Srđan Panić. Collections-C [online]. Accessed 5th-July-2021. URL: `https://github.com/srdja/Collections-C` [cited 5th July 2021].

53 Lawrence C Paulson. Isabelle [online]. Accessed 27th-November-2021. URL: `https://isabelle.in.tum.de`.

54 Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *IEEE/ACM International Conference on Automated Software Engineering*, 2010.

55 Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage*, 2013.

56 Andreas Rossberg. WebAssembly Reference Interpreter [online]. Accessed 3rd-December-2020. URL: `https://github.com/WebAssembly/spec/tree/master/interpreter` [cited 3rd December 2020].

57 Andreas Rossberg. WebAssembly Core Specification. Technical report, W3C, 2019. URL: `https://www.w3.org/TR/wasm-core-1`.

58 Sebastian Ruland, Malte Lochau, and Marie-Christine Jakobs. HybridTiger: Hybrid Model Checking and Domination-based Partitioning for Efficient Multi-Goal Test-Suite Generation (Competition Contribution). In *Fundamental Approaches to Software Engineering*, 2020.

59 Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. Digging into browser-based crypto mining. In *Internet Measurement Conference*, 2018.

60 José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *International Symposium on Principles and Practice of Declarative Programming*, 2018.

61 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 2019.

62 Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, 2010.

63 Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV*, 2006.

64 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes*, 2005.

65 Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Joint Meeting on Foundations of Software Engineering*, 2015.

66 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Conference on Annual Technical Conference*, 2012.

67 Amazon Web Services. AWS Encryption SDK for C. `https://github.com/aws/aws-encryption-sdk-c`. Accessed: 2021-09-08.

68 Quentin Stiévenart and Coen De Roover. Compositional Information Flow Analysis for WebAssembly Programs. In *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2020.

69 Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint tracking for WebAssembly. *arXiv preprint*, 2018.

70 The Coq Development Team. The Coq Proof Assistant [online]. Accessed 27th-November-2021. URL: `https://coq.inria.fr`.

71 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

72 W3C. W3C WebAssembly Core Specification [online]. Accessed 27th-November-2021. URL: `https://www.w3.org/TR/wasm-core-1`.

73 Dong Wang, Bo Jiang, and W. K. Chan. WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection, 2020. `arXiv:2007.15510`.

74 Conrad Watt. Mechanising and verifying the WebAssembly specification. In *ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.

75 Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. A program logic for first-order encapsulated WebAssembly. In Alastair F. Donaldson, editor, *European Conference on Object-Oriented Programming*, 2019.

76 Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two Mechanisations of WebAssembly 1.0. In *Formal Methods*, 2021.

77 Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven Secure Cryptography for the Web Ecosystem. *Proceeding of the ACM on Programming Languages*, 2019.

78 WebAssembly. WebAssembly FAQ [online]. Accessed: 29th-October-2021. URL: `https://webassembly.org/docs/faq`.

79 Shu-Hung You, Robert Bruce Findler, and Christos Dimoulas. Sound and complete concolic testing for higher-order functions. In *ESOP*, 2021.

Defining Corecursive Functions in Coq Using Approximations

Vlad Rusu ✉ iD
Inria, Lille, France

David Nowak ✉
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

We present two methods for defining corecursive functions that go beyond what is accepted by the builtin corecursion mechanisms of the Coq proof assistant. This gain in expressiveness is obtained by using a combination of axioms from Coq's standard library that, to our best knowledge, do not introduce inconsistencies but enable reasoning in standard mathematics. Both methods view corecursive functions as limits of sequences of approximations, and both are based on a property of productiveness that, intuitively, requires that for each input, an arbitrarily close approximation of the corresponding output is eventually obtained. The first method uses Coq's builtin corecursive mechanisms in a non-standard way, while the second method uses none of the mechanisms but redefines them. Both methods are implemented in Coq and are illustrated with examples.

2012 ACM Subject Classification Theory of computation → Functional constructs

Keywords and phrases corecursive function, productiveness, approximation, Coq proof assistant

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.12

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.2`

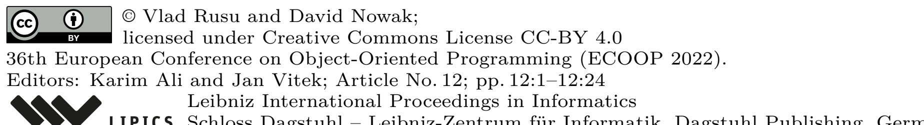
1 Introduction

Coq [1] is a proof assistant based on the Calculus of Inductive Constructions. Coinductive constructions (coinductive types, relations, and proofs, and corecursive functions) have been included in Coq's underlying theory [11]. However, these constructions are limited. Corecursive functions must conform to a syntactical *guardedness* criterion requiring that, up to standard reductions, calls to the function under definition occur directly under *constructors* of the coinductive representing the codomain of the function of interest. Such functions are total and by consequence are accepted by Coq.

The guardedness criterion is best illustrated by an example. Consider the set of streams S over a set A, which, intuitively, are infinite sequences of elements of A separated by the *constructor* $_ \cdot _$. The *head* (resp. the *tail*) of a stream s is the first element of s (resp. the stream obtained from s by removing its first element). Consider also a predicate p on A, and the following function *filter*, which takes a stream $s \in S$ as input and aims at producing as output a stream that contains the elements of s that satisfy p. Since its output is an (infinite) stream the function does not terminate.

$$\mathit{filter}\, s := \text{if } p(\mathit{head}\, s) \text{ then } (\mathit{head}\, s) \cdot (\mathit{filter}\, (\mathit{tail}\, s)) \text{ else } \mathit{filter}\, (\mathit{tail}\, s)$$

The first self-call to *filter* in the function's body falls directly under a call to the constructor $_ \cdot _$. It is therefore syntactically guarded by the constructor. In the second self-call, the constructor is not present; the second call is not syntactically guarded. Overall, the above function definition fails to satisfy the syntactical "guarded-by-constructors" criterion because of the second self-call.


© Vlad Rusu and David Nowak;
licensed under Creative Commons License CC-BY 4.0

To see why the syntactical guardedness criterion is important, consider a stream in $s \in S$ such that none of its elements satisfy p. Then, *filter* s is not a stream – none of the elements in the input are kept in the output. The unguarded call is responsible for this. Hence, *filter* is *not* a *total* function from S to S, and Coq's guardedness criteria rightfully reject it because Coq only accepts total functions.

However, by restricting its domain to *the set* S' *of streams* s' *such that infinitely many elements of* s' *satisfy* p, the *filter* function becomes a total function from S' to S. Intuitively, the guarded call, which copies an element in the input into the output, is called infinitely many times and thus produces an (infinite) stream. Such a function could, in principle, be accepted by Coq; however, Coq does not have automatic mechanisms to realize this. Its builtin syntactical criteria are automatic, sound (i.e. all functions that fulfill them are total), but restricted since they reject some total functions.

Let us take a closer look at the argument for the totality of *filter* restricted to S'. Consider an arbitrary stream $s' \in S'$. At the beginning, i.e., before *filter* s' starts computing, obviously, nothing is known about the value of output. This continues to be the case while the function processes successive elements of s' that do not satisfy p, because such elements are not kept in the output. However, *eventually*, an element of s', say, a, which does satisfy p, is encountered. It is kept in the output, which becomes $a \cdot (filter\,(tail\ \ldots))$, i.e., a stream about which *something* is known: its first element. Thus, one starts with a situation for which nothing is known about the output, and, *eventually*, the first element of the output becomes known. By repeating these observations one can see that, *eventually*, any finite prefix of the output becomes known. By viewing a finite prefix of a stream as an approximation of the stream in question, and by interpreting a longer prefix as a closer approximation of a stream than a shorter prefix, we can rephrase the argument for the totality of our function as: *for each input, an arbitrarily close approximation of the corresponding output is eventually produced.* This condition is called *productiveness*, and it is the condition that our function (and, in general, corecursive functions) needs to fulfill in order to be total. We note that in the literature about corecursive functions this condition (under various formulations) is well known, to the point that it has become folklore; but, to our best knowledge, it has not yet been formalized.

What is, then, the relation between guardedness and productiveness? To see this, consider the function $filter'\,s :=$ if $p(head\,s)$ then $(head\,s) \cdot (filter'\,(tail\,s))$ else $dummy \cdot filter'\,(tail\,s)$, in which the second self-call is now also guarded by the constructor _ · _ by having elements of the input that do not satisfy the predicate replaced by some dummy value in the output. The effect of this *guarded* definition is that for each input, the *next* call produces a closer approximation of the output. Since "next" is a particular case of "eventually", the overall effect of guardedness is to ensure productiveness and therefore totality, in a syntactical (thus, automatically checkable) and conservative way.

Contributions

In this paper we propose a formal definition of productiveness that captures the corresponding intuitive notion, and two methods for defining corecursive functions in which productiveness is a key ingredient. Essentially, productiveness restricts the manner in which a sequence of approximating functions converges to the function under definition, and the two methods offer two different ways for building the sequence of approximating functions. Both methods enable the definition of corecursive functions beyond what Coq accepts by default. Both methods have been implemented in Coq and are illustrated by examples. Their additional expressiveness is obtained thanks to axioms from Coq's standard library, which, to our best

knowledge, do not introduce inconsistencies. The difference between the methods lies in the amount of Coq builtin coinductive features they reuse: the first method reuses them extensively, while the second method uses none but redefines them.The Coq development is available at `https://project.inria.fr/ecoop2022/`.

The rest of the paper is organized as follows. A theoretical part (Sections 2-4) presents a formal notion of productiveness and our two corecursive function-definition methods in a language-agnostic manner. We emphasize that knowing Coq is not necessary for understanding the theory. Section 5 gives details of the Coq implementation that are not visible in the theory but are essential in the implementation, such as the combination of axioms imported from the standard library that enable reasoning in standard mathematics. Section 6 concludes and discusses related and future work.

2 A formal notion of productiveness

We start with some basic definitions used in the rest of the paper. Consider a set C and a partial order $\preceq$ on C. We denote by $\prec$ the relation defined by $t \prec t'$ iff $t \preceq t'$ and $t \neq t'$.

▶ **Definition 1.** *A sequence $(s_i)_{i\in\mathbb{N}}$ of elements of C is*

- increasing *whenever for all $i \in \mathbb{N}$, $s_i \preceq s_{i+1}$;*
- strictly increasing, *whenever for all $i \in \mathbb{N}$, $s_i \prec s_{i+1}$;*
- stabilizing to $c \in C$ *whenever there exist $m \in \mathbb{N}$ such that for all $i \geq m$, $s_i = c$, and* stabilizing *whenever it is stabilizing to some $c \in C$;*
- ascending *whenever it is increasing and non-stabilizing.*

▶ Remark. A sequence is ascending iff it is increasing and has a strictly increasing subsequence.

The following is one of the several existing definitions of a complete partial order (CPO) :

▶ **Definition 2.** *A CPO consists of a set C, a partial order $\preceq$ on C, and an element $\bot \in C$ satisfying $\forall t \in T$, $\bot \preceq t$, such that that any increasing sequence of elements of T has a least upper bound.*

We call the least upper bound of an increasing sequence $(s_n)_{n\in\mathbb{N}}$ the *limit* of the sequence, hereafter denoted by $lim[(s_n)_{n\in\mathbb{N}}]$.

▶ **Example 3.** Any set A can be organized as a CPO $(A \cup \{\bot_A\}, \preceq_A, \bot_A)$ by choosing some value $\bot_A \notin A$ and by defining $\preceq_A$ as the smallest relation on $A \cup \{\bot_A\}$ satisfying $\bot_A \preceq_A a$ for all $a \in A$ and $a' \preceq_A a'$ for all $a' \in A \cup \{\bot_A\}$. The properties of orders (reflexivity, anti-symmetry, transitivity) hold trivially. Any increasing sequence $(a_n)_{n\in\mathbb{N}}$ stabilizes to some $a \in A \cup \{\bot_A\}$, and the limit of the sequence is the value to which the sequence stabilizes. This CPO is called the *flat CPO of A*.

In the rest of the paper the maximal elements of a CPO with respect to its order shall play an important role: that of "well-defined corecursive values". This view is consistently held ahead in the paper.

The following definition is our formal notion of productiveness. It restricts the manner in which a sequence of functions "converges" to a given function.

▶ **Definition 4.** *Given a sequence of functions $(f_n)_{n\in\mathbb{N}}$ having the same domain D and codomain C, such that the codomain is organized as a CPO $(C, \preceq, \bot)$, we say that the sequence $(f_n)_{n\in\mathbb{N}}$* productively converges *whenever for all $x \in D$, the sequence $(f_n\, x)_{n\in\mathbb{N}}$ is increasing and its limit $lim[(f_n\, x)_{n\in\mathbb{N}}]$ is maximal w.r.t. the order $\preceq$. The* limit *of the sequence $(f_n)_{n\in\mathbb{N}}$ is by definition the function $f : D \to C$ such that for all $x \in D$, $f\, x = lim[(f_n\, x)_{n\in\mathbb{N}}]$. We call $(f_n)_{n\in\mathbb{N}}$ the sequence of approximating functions for the limit function f.*

▶ Remark. The image of the domain D by functions f constructed as in Definition 4 is included in the set of maximal elements of C, which, as said earlier, play the role of well-defined corecursive values. This justifies us calling "corecursive" the limits of productively converging sequences $(f_n)_{n \in \mathbb{N}}$.

▶ Remark. We now justify why Definition 4 captures the informal definition of productiveness. For each $x \in D$, the increasing sequence $(f_n\,x)_{n \in \mathbb{N}}$ is either stabilizing or non-stabilizing. The values $x \in D$ for which $(f_n\,x)_{n \in \mathbb{N}}$ stabilizes are inputs on which f terminates. The values $x \in D$ for which $(f_n\,x)_{n \in \mathbb{N}}$ does not stabilize are such that the increasing sequence $(f_n\,x)_{n \in \mathbb{N}}$ is *ascending*: it has a strictly increasing subsequence $(f_{n_i}\,x)_{i \in \mathbb{N}}$ such that for all $i \in \mathbb{N}$, $f_{n_i}\,x \prec f_{n_{i+1}}\,x$, i.e., $f_{n_i}\,x$ and $f_{n_{i+1}}\,x$ both are approximations of the sequence's limit $f\,x$, but $f_{n_{i+1}}\,x$ is a strictly *closer* approximation of $f\,x$ than $f_{n_i}\,x$. Thus, $(f_n\,x)_{n \in \mathbb{N}}$ produces, as n grows, arbitrarily close approximations of the output $f\,x$. This captures the intuition of productiveness: the ability to eventually produce, for each input, arbitrarily close (and, in case of termination, exact) approximations of the corresponding output.

The next two sections present two methods for obtaining CPOs and corecursive functions as limits of sequences of approximating functions. The first method reuses as much as possible Coq's builtin mechanisms for corecursion. The second one replaces these mechanisms by other constructions.

3 First method

In this approach the carrier set of the CPO being defined is the set of terms of a type coinductively defined by Coq and the limits of increasing sequences in the CPO are Coq builtin corecursive functions. The approximating sequences for the corecursive functions under definition use a functional for the function in question. We illustrate the approach by defining the filter function on streams.

3.1 CPOs as coinductive types

▶ **Example 5.** The set S of streams (a.k.a infinite lists) over a base set $A \cup \{\bot_A\}$ can be organized as a CPO as follows. First, the flat CPO $(A \cup \{\bot_A\}, \preceq_A, \bot_A)$ is built as in Example 3. Then, the set S is defined to be the set of terms of a certain coinductive type, which, conceptually, are built by applying the following rule a countably infinite number of times : $a \cdot s \in S$ whenever $a \in (A \cup \{\bot_A\})$ and $s \in S$. This simultaneously defines the *constructor* function $_ \cdot _ : (A \cup \{\bot_A\}) \times S \to S$.

Then, we define the constant stream $\bot \in S$ as the stream satisfying the equation $\bot = \bot_A \cdot \bot$. This is an example of a corecursive definition, which is accepted by Coq, since the occurrence of $\bot$ in the right-hand side is guarded by (a direct call to) the constructor $_ \cdot _$. On the set S we define the functions *head* and *tail* by $head(a \cdot s) = a$ and $tail(a \cdot s) = s$. We also define the nth element of a stream by induction: $nth\,0\,s = head\,s$ and $nth\,(n+1)\,s = nth\,n\,(tail\,s)$. Regarding the order relation $\preceq$, it is the relation on S defined "pointwise", by $s_1 \preceq s_2$ iff for all $n \in \mathbb{N}$, $nth\,n\,s_1 \preceq_A nth\,n\,s_2$.

Then, we define the limit $lim[(s_n)_{n \in N}]$ of an increasing sequence of streams $(s_n)_{n \in N}$ by $lim[(s_n)_{n \in N}] = (lim_A[(head\,s_n)_{n \in N}]) \cdot (lim[(tail\,s_n)_{n \in \mathbb{N}}])$. That is, the head of the limit of a sequence of streams is the limit (in $A \cup \{\bot_A\}$) of the heads of the streams in the sequence, and the tail of the limit is the limit (in S) of the tails of the streams in the sequence. This is another example of a corecursive function, and one that can be defined in Coq using the tool's builtin constructions for corecursion, because in the right-hand side

of $\mathit{lim}[(s_n)_{n\in N}] = (\mathit{lim}_A[(\mathit{head}\, s_n)_{n\in N}]) \cdot (\mathit{lim}[(\mathit{tail}\, s_n)_{n\in \mathbb{N}}])$ the call to *lim* is guarded by the constructor $_ \cdot _$. In order to prove that the defined limit is indeed the least upper bound one can, e.g., reduce that property to a "pointwise" one, i.e., first proving that for all $m \in \mathbb{N}$, $\mathit{nth}\, m\, (\mathit{lim}[(s_n)_{n\in N})] = (\mathit{lim}_A[(\mathit{nth}\, m\ s_n)_{n\in N}])$ and then using the fact that lim_A computes least upper bounds in the flat CPO of A. Finally, we note that the limits of ascending sequences of streams over $A \cup \{\bot_A\}$ are streams over A (i.e., they do not contain any $\bot_A$), and are maximal with respect to $\preceq$.

▶ Remark. As illustrated by the above example, the maximal elements in CPOs play the role of "well defined" corecursive values, since they do not contain $\bot$ subterms, themselves interpreted as "undefined". The ascending sequences "push away" $\bot$ subterms, to the effect that, in their limit, all such subterms have been eliminated. Since $\bot$ is interpreted as "undefined", terms containing $\bot$ are "partially defined", and "pushing $\bot$ away" amounts to producing "better defined" values.

▶ Remark. The ability to define a CPO using Coq's builtin mechanisms relies on the ability of those mechanisms to accept the definitions of limits as corecursive functions. This works for many interesting coinductive datatypes (streams, colists, possibly infinite binary trees, ...) but not in general. For coinductive datatypes that are mutually dependent with inductive datatypes, the limits may require corecursive functions that contain self-calls guarded not by constructors of the coinductive datatype, but by recursive functions on the inductive datatype. Such "improperly guarded" functions are rejected by Coq. A second method presented ahead in the paper deals with such difficult cases.

▶ Remark. The construction in Example 5 is not the only way to organize streams over a set A into a CPO. Another possibility is to define the set of streams S as the set obtained by applying the rules $\bot \in S$ and $a \cdot s \in S$ if $a \in A$ and $s \in S$ for a finite or a countably infinite number of times. In this definition $\bot$ is a constructor (unlike $\bot$ in Example 5 where it was a defined function). The order relation and the notion of limit are also slightly different. We chose the construction in Example 5 because it has fewer technical complications: for example, the *head* function is total in Example 5, but partial in the alternative construction, which makes it more complicated to define.

3.2 Approximating sequences using functionals

This method assumes a functional $F : (D \to C) \to D \to C$ for the function of interest. The functional may be obtained, e.g., from an attempt to define the function $f : D \to C$ of interest directly in Coq, via a statement of the form $f := F f$. It is, of course, assumed that the attempt failed – i.e., it failed the guardedness criteria – otherwise one would just define f directly in Coq.

▶ **Example 6.** Consider the set S of streams over $A \cup \{\bot_A\}$ as in Example 5 and assume a predicate $p : A \cup \{\bot_A\} \to \{\mathit{true}, \mathit{false}\}$ such that $p\, \bot_A = \mathit{false}$. Let D be the subset of S consisting of streams s over A, such that $p\, (\mathit{nth}\, n\, s) = \mathit{true}$ for infinitely many $n \in \mathbb{N}$. The following pseudocode statement is an attempt to define the *filter* function over D, which computes the substream of values satisfying p:

$$\mathit{filter}\, s := \text{if } p(\mathit{head}\, s) \text{ then } (\mathit{head}\, s) \cdot (\mathit{filter}\, (\mathit{tail}\, s)) \text{ else } \mathit{filter}\, (\mathit{tail}\, s)$$

Equivalently, the function could be defined by $\mathit{filter} := F\, \mathit{filter}$ where F is the pseudocode for the functional below, which takes a function as input and produces an (anonymous) function as output:

$$F\, f := \lambda\, s.\ \text{if } p(\mathit{head}\, s) \text{ then } (\mathit{head}\, s) \cdot (f\, (\mathit{tail}\, s)) \text{ else } f\, (\mathit{tail}\, s)$$

These definitions, when translated to Coq syntax, are rejected by the tool, because they fail the guardedness criterion: the call to *filter* in the "else" case is not guarded by the constructor $_ \cdot _$.

We show below an alternative way in which a functional F can be used for uniquely defining the function, say, f, of interest, while ensuring that the fixpoint equation $f = F\,f$ holds. Assume a CPO $(C, \preceq, \bot)$. We extend the order $\preceq$ from C to functions $D \to C$ by $f_1 \preceq f_2$ iff $f_1\,x \preceq f_2\,x$ for all $x \in D$.

▶ **Definition 7.** *A functional $F : (D \to C) \to D \to C$ is* increasing *if for all $f_1, f_2 : D \to C$, $f_1 \preceq f_2$ implies $F f_1 \preceq F f_2$.*

▶ **Example 8.** The functional F from the previous example is increasing. Indeed, consider two functions $f_1, f_2 : D \to S$, with D, S as in the example in question (in particular, S is organized as a CPO as in Example 5) and assume $f_1 \preceq f_2$. We have to show that $F\,f_1\,s \preceq F\,f_2\,s$ for all $s \in D$. If $p\,(head\,s) = true$ then $F\,f_1\,s = (head\,s) \cdot f_1\,(tail\,s)$ and $F\,f_2\,s = (head\,s) \cdot f_2\,(tail\,s)$; and $F\,f_1\,s \preceq F\,f_2\,s$ because $f_1 \preceq f_2$ implies in particular $f_1\,(tail\,s) \preceq f_2\,(tail\,s)$, and then $(head\,s) \cdot f_1\,(tail\,s) \preceq (head\,s) \cdot f_2\,(tail\,s)$ holds thanks to the definition of $\preceq$. If $p\,(head\,s) = false$ then $F\,f_1\,s = f_1\,(tail\,s)$, $F\,f_2\,s = f_2\,(tail\,s)$, and $F\,f_1\,s \preceq F\,f_2\,s$ is just $f_1\,(tail\,s) \preceq f_2\,(tail\,s)$, established above.

Assume again a CPO $(C, \preceq, \bot)$ and a functional $F : (D \to C) \to D \to C$. Let $\perp\!\!\!\perp : D \to C$ be the constant function such that $\perp\!\!\!\perp\ x = \bot$, for all $x \in D$, and let $F^n : (D \to C) \to D \to C$ be the functional inductively defined by $F^0 f = f$ and, for all $n \in \mathbb{N}$, $F^{n+1} f = F(F^n f)$.

▶ **Definition 9.** *A functional $F : (D \to C) \to D \to C$ is productive whenever it is increasing and the sequence of functions $(F^n \perp\!\!\!\perp)_{n\in\mathbb{N}}$ productively converges (cf. Definition 4).*

Calling *productive* a functional satisfying the above definition is justified by the fact that it generates a sequence of functions that productively converges. Its limit is characterized by the following theorem.

▶ **Theorem 10.** *If a functional F is productive then $lim[(F^n \perp\!\!\!\perp)_{n\in\mathbb{N}}]$ is the unique fixpoint of F.*

Proof. Let the type of the functional be $(D \to C) \to D \to C$. By Definition 4, for all $x \in D$, the sequence $(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}$ is increasing and its limit is maximal w.r.t. $\preceq$. Hence, for all $x \in D$, $lim[(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}]$ exists, and let $f : D \to C$ be defined by $f\,x = lim[(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}]$ for all $x \in D$.

We first show that f is a fixpoint of F, i.e., $f = Ff$, which amounts to proving that for all $x \in D, f\,x = F\,f\,x$. We fix an arbitrary $x \in D$. By definition of f, $f\,x$ is maximal, hence, in order to prove $f\,x = F\,f\,x$ it is enough to prove $f\,x \preceq F\,f\,x$. Moreover $f\,x$ is the least upper bound of $(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}$, hence, in order to show that $f\,x \preceq F\,f\,x$ it is enough to prove that $F\,f\,x$ is an upper bound of the sequence $(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}$. This is proved by case analysis: For $n = 0$, $F^0 \perp\!\!\!\perp\ x = \bot \preceq F\,f\,x$, and, for $n > 0$, we have that for all $y \in D$, $F^{n-1} \perp\!\!\!\perp\ y \preceq f\,y$ because $f\,y$ is an upper bound for the sequence $(F^k \perp\!\!\!\perp\ y)_{k\in\mathbb{N}}$, which, since F is increasing, implies that for all $y \in D$, $F\,(F^{n-1} \perp\!\!\!\perp)\,y = F^n \perp\!\!\!\perp\ y \preceq F\,f\,y$. Setting $y := x$ in the previous relation proves that $F\,f\,x$ is an upper bound of the sequence $(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}$ also in the case $n > 0$, and the proof of the fact that f is a fixpoint of F is completed.

Next, we show that f is the only fixpoint of F. Assume a solution f' of the fixpoint equation; we show $f = f'$. Note that it is enough to show $f \preceq f'$, i.e., $f\,x \preceq f'\,x$ for all $x \in D$, because from the latter by the maximality of $f\,x$ we obtain $f\,x = f'\,x$ for all $x \in D$, i.e., the desired $f = f'$.

Moreover, in order to prove that $f\,x \preceq f'\,x$ for all $x \in D$, it is enough to prove that $f'\,x$ is an upper bound for the sequence $(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}$, because by definition $f\,x$ is the least upper bound of the sequence. Hence, what we have to prove is that for all $n \in N$, (for all $x \in D$, $F^n \perp\!\!\!\perp\ x \preceq f'\,x$), which is done by induction on n. The base case $n = 0$ is trivial, as it amounts to showing that for all $x \in D$, $\perp \preceq f'\,x$. In the inductive step, we have the inductive hypothesis that for all $x \in D$, $F^n \perp\!\!\!\perp\ x \preceq f'\,x$. By using the fact that F is increasing we obtain that for all $x \in D$, $F^{n+1} \perp\!\!\!\perp\ x = F\,(F^n \perp\!\!\!\perp)\,x \preceq F\,f'\,x = f'\,x$, which proves the inductive step. That was what remained to prove; the proof of the theorem is complete. ◀

The productiveness condition is more convenient to establish via the following sufficient conditions.

▶ **Definition 11.** *A CPO $(C, \preceq, \perp)$ is a CPO+ if each ascending sequence has a maximal limit.*

▶ **Example 12.** Per the observation at the end of Example 5, the CPO of streams is a CPO+.

▶ **Lemma 13.** *Assume a CPO+ $(C, \preceq, \perp)$ having the set of maximal elements $K \subseteq C$, and a functional $F : (D \to C) \to D \to C$. Then, F is productive whenever it is increasing and, for all $x \in D$:*

- *either there exists $n \in \mathbb{N}$ such that $F^n \perp\!\!\!\perp\ x \in K$;*
- *or, for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ with $n < m$ such that $F^n \perp\!\!\!\perp\ x \prec F^m \perp\!\!\!\perp\ x$.*

Proof. By Definitions 4 and 9, we have to show that for all $x \in D$, the sequence $(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}$ has a limit in K. We first prove that the sequence is increasing, i.e., for all $n \in \mathbb{N}$, by induction on n. In the base case $n = 0$, we have, for each $x \in D$, $F^0 \perp\!\!\!\perp\ x = \perp\!\!\!\perp\ x = \perp \preceq F^1 \perp\!\!\!\perp\ x$, which settles this case. For the inductive step, we assume that for each $x \in D$, $F^n \perp \perp\ x \preceq F^{n+1} \perp\!\!\!\perp\ x$ and prove that, again for each $x \in D$, $F^{n+1} \perp\!\!\!\perp\ x \preceq F^{n+2} \perp\!\!\!\perp\ x$. We have $F^{n+1} \perp\!\!\!\perp\ x = F\,(F^n \perp\!\!\!\perp)\,x$ and since F is increasing, using the induction hypothesis $F\,(F^n \perp\!\!\!\perp)\,x \preceq F\,(F^{n+1} \perp\!\!\!\perp)\,x = F^{n+2} \perp\!\!\!\perp\ x$ holds for each $x \in D$, which proves the induction step and the fact that the sequence is increasing.

Hence, the sequence $(F^n \perp\!\!\!\perp\ x)_{n\in\mathbb{N}}$ has a limit; we just have to show the limit is in K.

- if there exists $n \in \mathbb{N}$ such that $F^n \perp\!\!\!\perp\ x \in K$, then, since the sequence is increasing and by definition of maximality, for all $m \geq n$, $F^m \perp\!\!\!\perp\ x = F^n \perp\!\!\!\perp\ x \in K$; and the limit is $F^n \perp\!\!\!\perp\ x \in K$ as required.
- if, for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ with $n < m$ such that $F^n \perp\!\!\!\perp\ x \prec F^m \perp\!\!\!\perp\ x$: we first note that each $F^n \perp\!\!\!\perp\ x$ must be in $C \setminus K$ (otherwise the hypothesis for this case is contradicted). Hence, the sequence has a strictly increasing subsequence, or, equivalently, the sequence is ascending. Since we have assumed that $(C, \preceq, \perp)$ is a CPO+ the limit of the sequence of interest is, again, in K. ◀

▶ **Example 14.** We prove using Lemma 13 that the functional $F : (D \to S) \to D \to S$ for the filter function from Example 6 is productive. We have already established that it is increasing and that the CPO $(S, \preceq, \perp)$ is a CPO+. We prove the condition at the second item the statement of Lemma 13, i.e., *for all $x \in D$ and $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ with $n < m$ such that $F^n \perp\!\!\!\perp\ x \prec F^m \perp\!\!\!\perp\ x$*, which amounts to finding a strictly increasing sequence of natural numbers $(n_i)_{i\in\mathbb{N}}$ such that $F^{n_i} \perp\!\!\!\perp\ x \prec F^{n_{i+1}} \perp\!\!\!\perp\ x$ for all $i \in \mathbb{N}$. This is where we use the fact that D is the set of streams x for which, given a predicate $p : (A \cup \{\perp_A\}) \to \{\mathit{true}, \mathit{false}\}$ on the base type of S with $p\,\perp_A = \mathit{false}$, it holds that $p\,(\mathit{nth}\,n\,x) = \mathit{true}$ for infinitely many $n \in \mathbb{N}$. We first prove by induction on n that $F^n \perp\!\!\!\perp = \mathit{ffilter}\,n$ where $\mathit{ffilter} = \lambda n.\lambda x.(\mathit{if}\ n = 0\ \mathit{then}\ \perp\ \mathit{else}\ (\mathit{if}\ p(\mathit{head}\,x)\ \mathit{then}\ (\mathit{head}\,x)\cdot(\mathit{ffilter}\,(n-1)\,(\mathit{tail}\,x))\ \mathit{else}\ \mathit{ffilter}\,(n-1)\,(\mathit{tail}\,x)))$ is a

recursive, "finite approximation" of the corecursive *filter* function that we are trying to define. Then, we notice that if $p(head\,x) = true$ then $\mathit{ffilter}\,(n+1)\,x = (head\,x)\cdot(\mathit{ffilter}\,n\,(tail\,x)$, and if $p(head\,x) = false$ then $\mathit{ffilter}\,(n+1)\,x = \mathit{ffilter}\,n\,(tail\,x)$. That is, for the positions in the sequence x where p holds, the output of *ffilter* "grows", and for the positions where p does not hold, the output of *ffilter* stays the same. Finally, for all $i \in \mathbb{N}$, let n_i be i-th position where p holds in x; this gives us the strictly increasing sequence $(n_i)_{i\in\mathbb{N}}$ such that $F^{n_i} \perp\!\!\!\perp x = \mathit{ffilter}\,n_i\,x \prec \mathit{ffilter}\,n_{i+1}\,x = F^{n_{i+1}} \perp\!\!\!\perp x$ for all $i \in \mathbb{N}$. Hence, using Lemma 13 we have established that the functional $F = \lambda f.\lambda s.$ *if* $p(head\,s)$ *then* $(head\,s)\cdot(f\,(tail\,s))$ *else* $f\,(tail\,s)$ is productive. Using Theorem 10 we obtain that F has a unique fixpoint; we call *filter* the fixpoint in question. The fixpoint equation states that $\mathit{filter}\,s =$ *if* $p(head\,s)$ *then* $(head\,s)\cdot(\mathit{filter}\,(tail\,s))$ *else* $\mathit{filter}\,(tail\,s)$ for all $x \in D$. We note that D being the set of streams having infinitely many positions satisfying the filtering predicate is essential: outside this domain the functional F is not productive and one cannot use Theorem 10 as above to define the filter function.

Summarizing, what we have obtained in the present section is a method by which corecursive functions can be defined in Coq – details about the Coq implementation are given in Section 5 – even when the functions are not directly accepted by Coq because they do not satisfy Coq's builtin criteria for corecursive definitions. A function defined using our approach is abstract (it involves limits of ascending sequences in a certain CPO), but is the unique one satisfying the equation induced by its functional. We use the term "validation" for the process by which one can gain confidence that a given definition is the adequate one; one can reasonably claim that uniquely satisfying its fixpoint equation is the best validation possible for a corecursive function.

Finally, we note that from the user's point of view, by using our approach one gets the same result that one would have gotten if Coq had directly accepted the corecursive definition. Our definitions are not executable because they use axioms – i.e., the term $\mathit{filter}\,s$ is not automatically reduced to the term *if* $p(head\,s)$ *then* $(head\,s)\cdot(\mathit{filter}\,(tail\,s))$ *else* $\mathit{filter}\,(tail\,s)$ by Coq – but, in order to avoid nontermination, such reductions are not performed in Coq-builtin corecursive definitions either: one still has to prove a fixpoint equation and manually perform, e.g., rewriting with it in order to reduce it.

4 Second method

When the technique presented for in the previous section fails, we need to replace Coq's builtin mechanisms for coinduction, which no longer fulfill their role, by other constructions.

4.1 CPOs built by completion

The main idea is to start from the "finite subset" of the intended CPO and from an order relation on the given subset, and to "complete" them with values that are the equivalence classes of ascending sequences, according to a certain equivalence relation. We illustrate the notions introduced in this section by giving an alternative construction of a CPO of streams, different from the construction based on Coq's builtin mechanisms seen in the previous section. We also show an example where the present construction of a CPO is essential because Coq's builtin mechanisms for coinduction fail.

▶ **Definition 15.** *Given a set C° and an order $\preceq^\circ$ on it, a* measure *on $(C^\circ, \preceq^\circ)$ is a function $\mu : C^\circ \to \mathbb{N}$ such that for all $x, y \in C^\circ$, $x \prec^\circ y$ implies $\mu\,x < \mu\,y$.*

That is, that the measure is compatible with the relation $\prec^\circ$. It is then easy to prove that a measure is also compatible with $\preceq^\circ$: $x \preceq^\circ y$ implies $\mu\, x \leq \mu\, y$.

▶ **Example 16.** Consider the set L of finite lists over a base set A, inductively defined by the rules $nil \in L$ and $a \cdot l \in L$ whenever $a \in A$ and $l \in L$. Define an order on L by $l_1 \preceq^L l_2$ iff l_1 is a prefix of l_2. Then, the function *length* mapping each list to its length is a measure on $(L, \preceq^L)$.

▶ Remark. For ascending sequences $(s_n)_{n\in\mathbb{N}}$, the sequence $(\mu\, s_n)_{n\in\mathbb{N}}$ is a sequence of natural numbers that tends to infinity. This is the main reason why we chose natural numbers as measure values.

▶ **Definition 17.** *Two sequences $(s_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}}$ of elements of C° are in the $\sim$ relation whenever for all $N \in \mathbb{N}$ there exist $n \in \mathbb{N}$ and $x \in C^\circ$ such that $x \preceq^\circ s_n$, $x \preceq^\circ s'_n$, and $\mu\, x \geq N$.*

A common predecessor of two elements is, in our approach, an "under-approximation" of the two elements. Thus, two sequences are in the $\sim$ relation whenever there there is a sequence of pointwise "under-approximations" of two sequences, whose measures tend to infinity. In some sense, the pointwise "difference" between the sequences intuitively tends to "nothing"[1]. In order to show that $\sim$ restricted to ascending sequences is an equivalence, the following property of an order is required:

▶ **Definition 18.** *An order $\sqsubseteq$ on a set A is weakly total whenever for all $a \in A$, the restriction of $\sqsubseteq$ to the set $\{a' \in A | a' \sqsubseteq a\}$ is total.*

▶ **Example 19.** The prefix order $\preceq^L$ on lists over a set A is weakly total: when l_1 and l_2 are both prefixes of a given list l then, if $length\, l_1 \leq length\, l_2$, $l_1 \preceq^L l_2$ holds, otherwise, $l_2 \preceq^l l_1$ holds. If A contains two elements $a_1 \neq a_2$, the order is not total, as $[a_1; a_2]$ and $[a_2; a_1]$ are incomparable.

▶ **Lemma 20.** *Assuming a set C° and a weakly total order $\preceq^\circ$ on C°, the restriction of the relation $\sim$ from Definition 17 to ascending sequences of elements of C° is an equivalence relation.*

Proof. For reflexivity, we use the fact that the sequence $(\mu\, s_n)_{n\in\mathbb{N}}$ of measures of an ascending sequence $(s_n)_{n\in\mathbb{N}}$ tends to infinity, hence, for each N there is $n \in \mathbb{N}$ such that $\mu\, s_n \geq N$, and we take $x := s_n$ in Definition 17 to show $(s_n)_{n\in\mathbb{N}} \sim (s_n)_{n\in\mathbb{N}}$. For symmetry, it is enough to note that Definition 17 is a symmetrical statement in $(s_n)_{n\in\mathbb{N}}$, $(s'_n)_{n\in\mathbb{N}}$. For transitivity assume $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}} \sim (s''_n)_{n\in\mathbb{N}}$. Fix an arbitrary $N \in \mathbb{N}$. By Definition 17, there exist $m, m' \in \mathbb{N}$ and $y, y' \in C^\circ$ such that $y \preceq^\circ s_m$, $y \preceq^\circ s'_m$, $y' \preceq^\circ s'_{m'}$, $y' \preceq^\circ s''_{m'}$, and $\mu\, y$, $\mu\, y' \geq N$. Since the sequences are increasing, we have $y, y' \preceq^\circ s'_{(max\, m\, m')}$ and since $\preceq^\circ$ is weakly total, $y \preceq^\circ y'$ or $y' \preceq^\circ y$. Assume $y \preceq^\circ y'$. Then, for the arbitrarily chosen N, we set $n := (max\, m\, m')$ and $x := y$ in Definition 17 and, since the sequences are increasing, we obtain $(s_n)_{n\in\mathbb{N}} \sim (s''_n)_{n\in\mathbb{N}}$. The other case ($y' \preceq^\circ y$) is similar. ◀

The next lemma gives a useful sufficient condition for the equivalence of ascending sequences.

[1] This intuition can be formalized using a notion of distance, thus turning C° into a metric space. We have tried but discarded that approach because is complicates matters (one now has an order, a distance, and a measure, which have to satisfy certain properties) without any other benefit that perhaps a better intuition for the notion of equivalence.

▶ **Lemma 21.** *Given two ascending sequences $(s_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}}$, if for all $k \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that $s_k \preceq s'_m$, then $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$.*

Proof. Fix an arbitrary $N \in \mathbb{N}$. Since $(s_n)_{n\in\mathbb{N}}$ is ascending, there exists $k \in \mathbb{N}$ such that $\mu\, s_k \geq N$. From the hypothesis we obtain $m \in \mathbb{N}$ such that $s_k \preceq^\circ s'_m$. Let $n := (max\, k\, m)$ and $x := s_k$. Since the sequences are increasing, $x \preceq s_n$, $x \preceq^\circ s'_n$, and from $\mu\, s_k \geq N$ we obtain $\mu\, x \geq N$. Hence, for all $N \in \mathbb{N}$ there are $n \in \mathbb{N}$, $x \in C^\circ$ such that $x \preceq^\circ s_n$, $x \preceq^\circ s'_n$, $\mu\, x \geq N$. By Def. 17, $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$. ◀

▶ Remark. The reverse implication in Lemma 21 does not hold in general: there exists sequences $(s_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}}$ such that for all $n, m \in \mathbb{N}$, s_n and s'_m are incomparable, yet $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$ because the sequences have in common another sequence that pointwise under-approximates them and whose measure tends to infinity, i.e., they obey Definition 17. The latter is the proper definition of equivalence: if instead we had taken *for all $k \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that $s_k \preceq s'_m$* as in Lemma 21 we would be distinguishing certain sequences – namely, those that have a common sequence of under-approximations whose sizes tend to infinity, yet are pointwise incomparable – that should not be distinguished, because pointwise the difference between them becomes “negligible”.

▶ **Definition 22.** *Assuming a set C° and a weakly total order relation $\preceq^\circ$ on C°, the* completion *of the set and its order to a set C and an order $\preceq$ on C are defined as follows:*

- *$C = C^\circ \cup K$, where K is the set of equivalence classes modulo $\sim$ of ascending sequences of elements in C°:*
- *$\preceq$ is the smallest relation on C satisfying*
  - *for all $x, y \in C^\circ$, $x \preceq y$ if $x \preceq^\circ y$;*
  - *for all $x, y \in K$, $x \preceq y$ if $x = y$;*
  - *for all $x \in C^\circ$ and $y \in K$, $x \preceq y$ if for all $(s_n)_{n\in\mathbb{N}} \in y$, there exists $m \in \mathbb{N}$ such that $x \preceq^\circ s_m$.*

This definition deserves a few comments. First, K is defined as equivalence classes of *ascending* sequences because, on the one hand, the sequences have to be increasing because they need to have limits – as we shall see, the set K will be a set of limits – and, on the other hand, they are non-stabilizing because if one sequence were stabilizing to a value, e.g., $v \in C^\circ$ then the limit (also v) of the sequence being also in K would imply a nonempty intersection of C° and K, which we wish to avoid. Second, the relation $\preceq$ is an order relation (this is established by Lemma 23 below). It is a conservative extension of $\preceq^\circ$, and elements in K are in the order iff they are equal. Combined with the fact that there is no situation in which $x \preceq y$ for $x \in K$ and $y \in C^\circ$, we obtain that the elements in K are maximal w.r.t. $\preceq$. Like in the case of the CPO of streams in an earlier example, the maximal elements play the role of “well-defined corecursive values”. Finally, the third case defining the relation $\preceq$ requires an explanation. An element x (in C°) is in the order with an equivalence class y of ascending sequences (in K) whenever each sequence in the class “overtakes” x at some position $m \in \mathbb{N}$ according to the base relation $\preceq^\circ$. Combined with the fact that $(s_n)_{n\in\mathbb{N}}$ is increasing, this implies that the sequence overtakes x for all positions $n \geq m$. Several results hereafter (Lemma 24, Theorem 26, Theorem 32) critically depend on the proposed definition of the $\preceq$ relation.

▶ **Lemma 23.** *Assume a measure μ on $(C^\circ, \preceq^\circ)$ like in Definition 15, with $\preceq^\circ$ a weakly total order. Then, with C and $\preceq$ being the completions of C° and $\preceq^\circ$ respectively, given in Definition 22, the relation $\preceq$ on C is an order.*

Proof. Reflexivity is trivial since $\preceq$ amounts to $\preceq^\circ$ on C° and to equality on K, both of which are reflexive. For anti-symmetry, we note that it reduces to the anti-symmetry of $\preceq^\circ$, because the nontrivial remaining case has the form "$x \preceq y$ and $y \preceq x$ for $x \in C^\circ$ and $y \in K$ implies $x = y$", which holds because its premise $y \preceq x$ is impossible. Let us now consider transitivity, thus, $x \preceq y$ and $y \preceq z$. There are only four possibilities when those relations can hold:

1. $x, y, z \in C^\circ$, in which case the transitivity of $\preceq$ reduces to that of $\preceq^\circ$;
2. $x, y \in C^\circ$ and $z \in K$, which implies $x \preceq^\circ y$ and, given the definition of $y \preceq z$ for x an element and z a equivalence class of ascending sequences, from the fact that any sequence in z overtakes y at some position, we obtain thanks to $x \preceq^\circ y$ that the sequence also overtakes x at the same position, which implies $x \preceq z$ and settles this case;
3. $x \in C^\circ$ and $y, z \in K$: then, $y \preceq z$ implies $y = z$, and transitivity follows easily;
4. $x, y, z \in K$, in which case the transitivity of $\preceq$ follows from that of equality. ◀

The following lemma gives a useful alternative definition for the order $\preceq$ in a particular case.

▶ **Lemma 24.** *For all $x \in C^\circ$ and ascending sequences $(s_n)_{n\in\mathbb{N}}$ of elements of C°, $x \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ iff there exists $m \in \mathbb{N}$ such that $x \preceq s_m$.*

Proof. By Definition 22, $x \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ means: for all $(s'_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$, there exists $m \in \mathbb{N}$ such that $x \preceq^\circ s'_n$. The "only if" direction is trivial since obviously $(s_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$. We thus focus on the "if" direction. By hypothesis, there exists $m \in \mathbb{N}$ such that $x \preceq^\circ s_m$. Choose an arbitrary $(s'_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$, i.e., $(s'_n)_{n\in\mathbb{N}} \sim (s_n)_{n\in\mathbb{N}}$. By Definition 17, there exists $x' \in C^\circ$ and $m' \in \mathbb{N}$ such that $x' \preceq^\circ s_{m'}$, $x' \preceq^\circ s'_{m'}$ and $\mu\, x' > \mu\, x$. Since the sequences are increasing, we obtain $x, x' \preceq^\circ s_{(max\, m,\, m')}$. From the latter and the weak totality of $\preceq^\circ$ we obtain $x \preceq^\circ x'$ or $x' \preceq^\circ x$. But $x' \preceq^\circ x$ contradicts the established $\mu\, x' > \mu\, x$. Hence, $x \preceq^\circ x'$ and then $x \preceq^\circ s'_{m'}$ follows from $x' \preceq^\circ s'_{m'}$ by transitivity. Summarizing, for the arbitrarily chosen sequence $(s'_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$ we found $m' \in \mathbb{N}$ such that $x \preceq^\circ s'_{m'}$ř. But this is $x \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ by definition; which proves the lemma. ◀

▶ **Definition 25.** *Given a set C° and weakly total order $\preceq^\circ$ on C°, consider the completion of C° to C and of $\preceq^\circ$ to $\preceq$ as in Definition 22. For an increasing sequence $(s_n)_{n\in\mathbb{N}}$ of elements of C, we define $lim[(s_n)_{n\in\mathbb{N}}]$ as follows:*

- *if the sequence stabilizes at a value, say, $v \in C$, then $lim[(s_n)_{n\in\mathbb{N}}] = v$;*
- *otherwise, the sequence does not stabilize, which implies that for all $n \in \mathbb{N}$, $s_n \in C^\circ$, and we define $lim[(s_n)_{n\in\mathbb{N}}] = [(s_n)_{n\in\mathbb{N}}]_\sim$, i.e., the equivalence class of the sequence w.r.t. the relation $\sim$.*

Note that in the second case of the above definition it is essential that the ascending sequence $(s_n)_{n\in\mathbb{N}}$ be composed of elements of C° because $\sim$ is only an equivalence for such sequences.

▶ **Theorem 26.** *Assume a measure on $(C^\circ, \preceq^\circ)$ like in Definition 15, with $\preceq^\circ$ a weakly total order. Then, with C and $\preceq$ being the completions of C° and $\preceq^\circ$ respectively, given in Definition 22, and with the limits of increasing sequences introduced in Definition 25, the triple $(C, \preceq, \bot)$ is a CPO.*

Proof. In order to prove the theorem we have to prove that the limits of increasing sequences proposed in Definition 25 are least upper bounds. Consider an increasing sequence $(s_n)_{n\in\mathbb{N}}$ of elements of C.

- if the sequence stabilizes to some value $v \in C$ then the proposed limit v is an upper bound for the (increasing) sequence. To show that it is the least such bound, assume another upper bound w; then, in particular, $v \preceq w$ because v is an element of the sequence.

- if the sequence does not stabilize then it is ascending, and as already observed before, $s_n \in C^\circ$ for all $n \in \mathbb{N}$, and the proposed limit is the equivalence class $[(s_n)_{n\in\mathbb{N}}]_\sim$.
  - We first show that $[(s_n)_{n\in\mathbb{N}}]_\sim$ is an upper bound for $(s_n)_{n\in\mathbb{N}}$: $s_k \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ for all $k \in \mathbb{N}$. We apply Lemma 24 with $x := s_k$: there exists $m := k$ such $s_k \preceq s_m$, which implies $s_k \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$.
  - Then, we show that $[(s_n)_{n\in\mathbb{N}}]_\sim$ is the least upper bound for $(s_n)_{n\in\mathbb{N}}$. Assume any upper bound $w \in C$, thus, $s_k \preceq w$ for all $k \in \mathbb{N}$. Suppose first that $w \in C^\circ$. Since $(s_n)_{n\in\mathbb{N}}$ is ascending, it has a strictly increasing subsequence $(s_{n_i})_{i\in\mathbb{N}}$. Now, w is also an upper bound for the subsequence, hence, $s_{n_i} \preceq w$ for all $i \in \mathbb{N}$, and due to the properties of the measure, $\mu\, s_{n_i} \preceq \mu\, w$ for all $i \in \mathbb{N}$. But this is impossible, since the sequence of measures of a strictly increasing sequence is a strictly increasing sequence of natural numbers, which tends to infinity. Hence, $w \in C^\circ$ is impossible. It follows that $w \in K$, i.e., $w = [(s'_n)_{n\in\mathbb{N}}]_\sim$ for some ascending sequence $(s'_n)_{n\in\mathbb{N}}$. From our hypothesis $s_k \preceq w$ for all $k \in \mathbb{N}$, we obtain that for all $k \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that $s_k \preceq^\circ s'_m$. Using Lemma 21, $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$, i.e., $[(s_n)_{n\in\mathbb{N}}]_\sim = [(s'_n)_{n\in\mathbb{N}}]_\sim = w$ and in particular $[(s_n)_{n\in\mathbb{N}}]_\sim \preceq w$. Since the upper bound w was chosen arbitrarily, we have proved that $[(s_n)_{n\in\mathbb{N}}]_\sim$ is the least upper bound for $(s_n)_{n\in\mathbb{N}}$. The proof of the theorem is complete. ◀

► **Example 27.** Going back to the example of finite lists, their prefix order, and the measure defined by lengths of lists, the constructions in this section enable us to build a CPO of lists and streams. The streams are not defined by Coq corecursive functions (as in the earlier construction in Section 3) but by equivalence classes of ascending sequences of lists. One important difference in practice is that, unlike the approach in Section 3, the constructor $_ \cdot _$ is not directly available for streams, and the functions *head* and *tail* do not have simple definitions. All three functions can be defined, and the standard relations between them can be proved, with some effort; but having them readily available as in the approach from Section 3 is preferable. We now give an example where that approach fails.

► **Example 28.** The set T of Rose trees over a set A is coinductively definable in Coq by the rules $\bot \in T$ and $\mathit{tree}\, a\, l \in T$ whenever $a \in A$ and l is a list over T. Note the mixture of coinduction and induction: the trees are defined coinductively, but their definition relies on inductively defined lists.

When $t = \mathit{tree}\, a\, l$ we define $\mathit{label}\, t = a$ and $\mathit{forest}\, t = l$; when $t = \bot$ we define $\mathit{label}\, t = \bot_A$ (the least element in the flat CPO of A) and let $\mathit{forest}\, t$ be the singleton $[\bot_A]$. Assuming an order $\preceq^T$ on T, the limit of an increasing sequence $(t_n)_{n\in\mathbb{N}}$ of Rose trees would naturally be defined as $\mathit{lim}[(t_n)_{n\in\mathbb{N}}] = \bot$ if $t_n = \bot$ for all $n \in \mathbb{N}$ and $\mathit{lim}[(t_n)_{n\in\mathbb{N}}] = \mathit{tree}\,(\mathit{lim}_A[(\mathit{label}\, t_n)_{n\in\mathbb{N}}])\,(\mathit{map}\, \mathit{lim}\,(\mathit{forest}\, t_n)_{n\in\mathbb{N}})$ otherwise. This corecursive definition of limits is not guarded by constructors, since the corecursive call to *lim* occurs under the *map* function, which is not a constructor but a defined function. Hence, the definition of limits of increasing sequences of Rose trees is rejected by Coq, and without limits there is no CPO.

► **Example 29.** One can define and organize the set $T = F \cup R$, with F the set of finite trees and R that of Rose trees, in a CPO using the approach described in this section. We first define the finite trees F over A inductively, by the rules $\bot \in F$ and $\mathit{tree}\, a\, l \in F$ whenever $a \in A$ and l is a list over F. The measure function is the tree's height, recursively defined by $\mathit{height}\, \bot = 0$ and $\mathit{height}\,(\mathit{tree}\, a\, l) = 1 + \mathit{max}\,(\mathit{map}\, \mathit{height}\, l)$ where *max* computes the maximum value in a list of natural numbers. The order relation is based on the following recursive function, whose effect is to "cut" a given finite tree t at given depth n:

$cut = \lambda n.\lambda t.\mathit{if}\ n = 0\ \mathit{or}\ t = \bot\ \mathit{then} \bot\ \mathit{else}\ \mathit{tree}\,(\mathit{label}\,t)\,(\mathit{map}\,(\mathit{cut}\,(n-1))\,(\mathit{forest}\,t))$; we then define the order relation $\preceq^F$ by $t_1 \preceq^F t_2$ whenever $t_1 = \mathit{cut}(\mathit{height}\,t_1)\,t_2$. The set R of Rose trees consists of equivalence classes of ascending sequences of finite trees. We have proved in Coq that all the requirements presented earlier in this section for obtaining a CPO for $T = F \cup R$ hold.

By contrast, a perhaps more natural definition of the "prefix order" $\preceq'^F$ by $t_1 \preceq'^F t_2$ *whenever* $t_1 = \bot$ *or* $t_1 = \mathit{tree}\,a\,l_1$, $t_2 = \mathit{tree}\,a\,l_2$, $\mathit{length}\,l_1 = \mathit{length}\,l_2$ *and for all* $n < \mathit{length}\,l_1$, $\mathit{nth}\,n\,l_1 \preceq'^F \mathit{nth}\,n\,l_2$ fails to meet the critical weak totality requirement (Definition 18). Indeed, e.g., for $t', t'' \neq \bot$, $t = \mathit{tree}\,a\,[t', t'']$, $t_1 = \mathit{tree}\,a\,[t', \bot]$ and $t_2 = \mathit{tree}\,a\,[\bot, t'']$ satisfy $t_1 \preceq'^F t$ and $t_2 \preceq'^F t$, yet t_1 and t_2 are incomparable. Without weak totality there is no sequence equivalence and ultimately no CPO[2].

4.2 Approximating sequences without functionals

In Section 3.2 the approximating sequence $(f_n)_{n\in\mathbb{N}}$ for defining a function was defined using a functional, which used functions over streams (such as the constructor $_\cdot_$) that were readily available in Coq, due to the fact that the CPO for streams had been defined as a builtin Coq coinductive type. However, in the case of CPOs built by completion, such constructors are no longer available. One can try to replace them by defined functions, but this may turn out to be excessively difficult. For instance, in the CPO of Example 29, extending the constructor *tree* from finite trees F to a fully defined function $\mathit{tree} : A \to \mathit{list}\,T \to T$, with $T = F \cup R$, is difficult: each of the trees in its second argument of type $\mathit{list}\,T$ may be a finite tree in F or a Rose tree in R – an equivalence class of ascending sequences of elements in F. Even when all elements in the list are equivalence classes, it is not clear how the result – again, an equivalence class of ascending sequence of elements in F – can be built.

Hence, we have to make do without constructors or defined functions replacing them. This severely limits the functionals that one may write, making the functional-based definition of corecursive functions from Section 3.2 essentially useless. Example 31 below illustrates this issue. In this section we present an approach that does not require a functional, but does require a "finite version" f° of the corecursive function f under definition, which moreover has to satisfy a productiveness requirement.

▶ **Definition 30.** *Assume two CPOs* $(D, \preceq_D, \bot_D)$ *and* $(C \preceq_C, \bot_C)$ *defined as in Theorem 26, thus, their base sets are decomposed as* $C = C^\circ \cup K_C$ *and* $D = D^\circ \cup K_D$*. Then, a function* $f^\circ : D^\circ \to C^\circ$ *is* productive *whenever, for all increasing sequences* $(x_n)_{n\in\mathbb{N}}$ *of elements in* D° *that have a limit in* K_D*, the sequence* $(f^\circ x_n)_{n\in\mathbb{N}}$ *of elements in* C° *is also increasing and has a limit in* K_C*.*

▶ Remark. Definition 30 of a productive function implies the function is also increasing. It also implies that the function maps ascending sequences to ascending sequences. Calling such a function *productive* is justified by the fact that it generates a sequence of functions that productively converges according to Definition 4. This sequence is built as follows: for all $x \in K_D$, choose an arbitrary ascending sequence $(x_n)_{n\in\mathbb{N}} \in x$; and set $(f_n\,x) = (f^\circ x_n)$ for all $n \in \mathbb{N}$. Then, $(f_n)_{n\in\mathbb{N}}$ productively converges according to Definition 4: for all $x \in K_D$, $(f_n\,x)_{n\in\mathbb{N}}$ is increasing and its limit is in K_C.

[2] Our earlier attempt with metric spaces also required a weakly total order for obtaining a proper notion of distance.

▶ **Example 31.** In the CPO of finite and Rose trees from Example 29, the sets C° and D° from the above definition are both the set F of finite trees. Consider the following recursive endofunction of F: $mirror^\circ = \lambda t. if\, t = \bot\ then\ \bot\ else\ tree\,(label\, t)(map\, mirror^\circ\,(rev\,(forest\, t)))$, where *rev* is the function that computes the reverse of a list. As its name indicates, the function computes the "mirror image" of finite trees. We have defined this function in Coq and have proved that it is productive according to Definition 30, using the fact that the $mirror^\circ$ function preserves the *height* of its argument.

Note how the functional for $mirror^\circ$: $\lambda\phi\, t. if\, t = \bot\ then\ \bot\ else\ tree\,(label\, t)(map\,\phi\,(rev\,(forest\, t)))$ uses the constructor *tree*. Writing the corresponding functional for a full *mirror* function for both finite and Rose trees would require a corresponding defined function $tree : A \to list\, T \to T$. As stated above, such a function is hard to define. Hence our alternative solution avoiding these issues.

The following theorem states that productive functions map equivalent ascending sequences in their domain to equivalent ascending sequences in their codomain.

▶ **Theorem 32.** *In the context of Definition 30, let $\sim_D$ denote the equivalence relation on ascending sequences in the CPO $(D, \preceq_D, \bot_D)$ (cf. Definition 17, Lemma 20). Let $\sim_C$ denote the corresponding equivalence in $(C \preceq_C, \bot_C)$. Then, for any pair of equivalent ascending sequences $(s_n)_{n\in\mathbb{N}} \sim_D (s'_n)_{n\in\mathbb{N}}$ and any productive function $f^\circ : D^\circ \to C^\circ$, we have the equivalence $(f^\circ s_n)_{n\in\mathbb{N}} \sim_C (f^\circ s'_n)_{n\in\mathbb{N}}$.*

Proof. By Definition 22 and Theorem 26 the equivalence class $[(s_n)_{n\in\mathbb{N}}]_{\sim_D}$ is the least upper bound of $(s_n)_{n\in\mathbb{N}}$ and the equivalence class $[(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$ is the least upper bound of $(s'_n)_{n\in\mathbb{N}}$. Since the two sequences are equivalent, we have the equality $[(s_n)_{n\in\mathbb{N}}]_{\sim_D} = [(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$. In particular, it follows that $[(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$ is an upper bound for $(s_n)_{n\in\mathbb{N}}$, thus for all $n \in \mathbb{N}$, $s_n \preceq_D [(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$ and by Lemma 24, (i): *for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ such that $s_n \preceq^\circ_D s'_m$*. Since f° is productive, it is also increasing, and thus from (i) we obtain (ii): *for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ such that $f^\circ s_n \preceq^\circ_C f^\circ s'_m$*. Using Lemma 21 we obtain $(f^\circ s_m)_{m\in\mathbb{N}} \sim_C (f^\circ s'_m)_{m\in\mathbb{N}}$, which proves the lemma. ◀

The above result enables us to define functions $f : K_D \to C$ as limits of approximating sequences $(f_n : K_D \to C)_{n\in\mathbb{N}}$. The definition of each of the functions f_n below depends on an arbitrary choice for a representative in its argument (which is an equivalence class), however, thanks to the above theorem, the limit (i.e. the defined function f) does not depend on that choice. The functions are built as in the Remark following Definition 30: for all $x \in K_D$, choose an arbitrary ascending sequence $(x_n)_{n\in\mathbb{N}} \in x$; and set $(f_n\, x) = (f^\circ x_n)$ for all $n \in \mathbb{N}$. We have observed that $(f_n)_{n\in\mathbb{N}}$ productively converges according to Definition 4; its limit is $[(f^\circ x_n)_{\in\mathbb{N}}]_{\sim_C} \in K_C$, which, by above theorem, is independent of the choice for $(x_n)_{n\in\mathbb{N}} \in x$. Hence, the limit $f := \lambda x. lim[(f_n\, x)_{n\in\mathbb{N}}]$ is also independent on the initial choice.

Of course, the natural question that arises regards validation: do the functions thus defined match the intention of the user? Unlike the case of functional-based corecursive functions in an earlier section, we do not have a functional and a fixpoint equation as validation mechanisms. A certain degree of confidence in the definition of f is already obtained from the (assumed) confidence in f° – as we have $f([(x_n)_{n\in\mathbb{N}}]_{\sim_D}) = [(f^\circ x_n)_{n\in\mathbb{N}}]_{\sim_C}$ – and from the independence of choice of representative from Theorem 32. The confidence can be improved by proving properties of f that the user expects.

▶ **Example 33.** By applying the above process to the $mirror^\circ$ function from Example 31, and noting that in this case $K_D = K_C = R$ (the set of Rose trees), we obtain a well-defined function $mirror : R \to R$. To increase confidence in this function we prove that the *mirror*

function "reverses" positions in Rose trees. A position p is a finite sequence of natural numbers, and in a given tree t it indicates the label (in the set $A \cup \bot_A$, if we consider trees over A) obtained by navigating in the tree, starting from the root and choosing children indexed by the successive numbers in p. Let $pos\,p\,t$ be the label in question (or $\bot_A$, since the position may "overflow"). A function pos_rev is also defined, which like pos takes a position p and a tree t and navigates the tree from root to children, but unlike pos, the children are chosen "backwards" (counting back starting from the last child) instead of forwards. We have then proved that for all positions p and trees t (finite or Rose), $pos\,p\,(mirror\,t) = pos_rev\,p\,t$, meaning that, intuitively, $mirror$ "reverses" all positions in the tree. The proofs were performed by first defining finite versions pos° and pos_rev° for the new functions involved, then proving $pos^\circ p\,(mirror^\circ f) = pos_rev^\circ p\,f$ for finite trees f, and finally proving that the corresponding property on Rose trees reduces to that on finite trees whose height is large enough (here, larger the length of the list p). The Coq proofs are available in the companion Coq development.

5 Implementation

The corecursive function-definition methods presented in Sections 2–4 have been implemented in the Coq proof assistant. The implementation has two motivations. The first one is ensuring that the results are sound, i.e., no case has been forgotten in a proof, and no assumption was left implicit. This is a standard motivation for using a proof assistant. The second motivation aims at providing Coq itself with stronger mechanisms for corecursive definitions than the builtin ones available in the tool. This is achieved at the cost of assuming several axioms from Coq's standard library; we state which axioms were used, where, and for what purpose. To our best knowledge the combination of axioms we imported from the standard library does not introduce inconsistencies (cf. [8, Chapter 12]).

Understanding the rest of this section requires knowledge about Coq's inductive and coinductive types, recursive and corecursive definitions, and its module system.

5.1 Sequences

Some notions are used by both methods. The main concept is that of sequences over a given type, encoded as functions from the natural numbers to the type in question. The fact that an element belongs to a sequence (parameterized by a given type) is also defined using an existential quantifier.

```
Definition Seq {A:Type}:Type := nat -> A
Definition sin{A:Type}(a:A)(q: Seq(A:=A)):Prop := exists i, a = q i.
```

Then, given a relation `R` (i.e., a binary predicate, of type `A->A->Prop`), the various kinds of sequences from Definition 1 (increasing, strictly increasing, stabilizing, ascending) are defined. Next, the fact that a value `lub_val` is the least upper bound (w.r.t. a relation `R`) of a sequence `q` is defined as

```
Definition lub{A:Type} (R:A-> A-> Prop) (lub_val: A) (q:Seq(A:=A)) :=
(forall a, sin a q -> R a lub_val) /\
(forall lub_val',(forall a,sin a q-> R a lub_val')-> R lub_val lub_val').
```

The definition of `lub` is only relevant for order relations `R`, and will only be used for such relations. For the first method these definitions are enough. The second method requires the property noted in the Remark following Definition 1: if `R` is an order, then a sequence is ascending if and only if it is increasing and has a strictly increasing sequence. This one-line property required quite a few intermediary lemmas in order to be formally proved, using classical logic and the following axiom:

```
Axiom constructive_indefinite_description: forall (A:Type) (P:A->Prop),
(exists x, P x) -> {x:A | P x}.
```

This axiom, from Coq's standard library, enables one to "choose" an element P satisfying a predicate P just based on the knowledge that P is satisfiable. In informal mathematical reasoning this is often implicitly assumed. In a Coq formal development, however, it has to be explicitly assumed. Here we use it in order to turn a total relation into a function having the relation in question as its graph:

```
Lemma functional_choice: forall (A B:Type) (R:A->B->Prop),
(forall x:A,exists y:B, R x y)->(exists f:A->B,forall x:A, R x (f x)).
```

The `constructive_indefinite_description` axiom occurs several times in the Coq development.

▶ Remark. We have not formalized Definition 4 of productive convergence of sequences of functions. That definition is useful in the paper for a unified presentation of the two methods and for giving the intuitive notion of productiveness a mathematical meaning. In Coq these motivations do not apply.

5.2 First method

This method reuses Coq's builtin coinduction mechanisms for organizing coinductive types as CPOs.

5.2.1 Stream CPO

The Coq definition for the stream CPO closely follows the approach outlined in Example 5. First, the flat CPO over a given type A (cf. Example 3) is encoded using Coq's `option` type. A relation `leo` on this type is also defined, which we prove to be an order relation, having `None` as the bottom element:

```
Inductive option(A:Type): Type :=  None: option A | Some: A -> option A.
Inductive leo{A:Type} : option A -> option A -> Prop :=
|leo_none: forall a, leo None a
|leo_some: forall a, leo (Some a) (Some a)
```

In Example 3, `None` was denoted by $\bot_A$ and `leo` was denoted by the infix symbol $_ \preceq_A _$. Then, a lemma states that for each increasing sequence in the `leo` order, there exists a least upper bound:

```
Lemma leo_lub{A:Type} :
forall (q:Seq (A:=option A)),increasing leo q-> exists b,lub leo b q.
```

The least upper bound of a sequence is obtained using `constructive_indefinite_description`:

```
Definition limF{A:Type}(q:Seq(A:=option A))(H:increasing leo q):=
  constructive_indefinite_description _ (leo_lub q H).
```

Next, a stream over a type T is obtained by applying the constructor `scons` to an element in T and another stream over T. The stream `bot`, which is an infinite repetition of `None`, is also defined.

```
CoInductive Stream{T:Type} :=  scons : T -> Stream -> Stream.
CoFixpoint bot{T:Type}: Stream(T:=T) :=  scons None bot.
```

In Example 5 the constructor `scons` is denoted by an infix operation $_ \cdot _$ and `bot` is denoted by $\bot$. The head (`hd`) and tail (`tl`) of a stream are also defined, in the expected manner.

Next comes the order relation on streams. In Example 5 the order $_ \preceq _$ was defined pointwise. We here give an alternative, coinductive definition, and prove that the two definitions are equivalent.

```
CoInductive les{T:Type} :Stream(T:=T)-> Stream (T:=T)-> Prop :=
les_def:forall a b s s', leo a b-> les s s'-> les(scons a s)(scons b s').
```

Next, the limit of an increasing sequence of streams over the flat CPO of a given set is defined by:

```
CoFixpoint lim{A:Type}(q:Seq(A:=Stream(T:=option A)))(H:increasing les q)
:= scons
 (proj1_sig(limF(fun n => hd(q n))(increasing_smap_hd q H)))
 (lim(fun n => tl(q n))(increasing_smap_tl q H)).
```

The function is parameterized by base type `A` of the underlying flat CPO. The type of the argument `q` is a sequence of streams over the flat order of `A`. The function takes a second argument: a proof that the sequence is increasing w.r.t. the order `les` of streams. The function returns a stream, built with `scons`, whose head is the limit (`limF`, in the flat CPO) of a stream that consists in mapping the head of streams to the sequence `q`, and whose tail is the (corecursively called) limit of the sequence of streams obtained by mapping the tail of streams to the sequence `q`. There are also some proof terms being used for ensuring that the various sequences whose limits are being invoked are increasing. Finally, we prove that the proposed limit is the actual least upper bound of an increasing sequence:

```
Lemma lim_lub{A:Type}(q:Seq(A:=Stream(T:=option A)))(H:increasing les q):
lub les (lim q H) q.
```

We also formalize the main artifact in the first method for corecursive function definition – Theorem 10, which says that a productive functional has a unique fixpoint. For productiveness we use the more convenient sufficient conditions given by Lemma 13. These conditions are placed in a Coq *module type*, which can be seen as an interface that other modules need to implement in order to benefit from the results implied by the conditions (here, the function definition method embodied in Theorem 10).

5.2.2 The filter function on streams

The proposed functional for the filter function for streams over a type `A` is written as follows:

```
Definition Filter(f:S->Stream(T:=option A))(s: S): Stream(T:=option A):=
if P (head s) then
scons (head s) (f (tail s))
else f (tail s).
```

where `S:= {s:Stream(T:=option A)|forall n,exists m,n<=m` $\wedge$ `P(nth m q)=true}` is the subtype of streams that have an infinite number of elements satisfying the filtering predicate `P: option A-> bool`, and `head`, `tail` are the restrictions of the `hd`, resp. `tl` functions on streams to the subtype `S`. We prove the conditions in Lemma 13, which enables us to use the Coq formalization of Theorem 10 and to define a function `filter` satisfying the two following theorems:

```
Theorem filter_fix: forall  s, bisim (filter s)(Filter (filter s))
Theorem filter_fix_unique: forall f,(forall s,bisim (f s) (Filter f s))->
 forall s,bisim  (filter s)(f s).
```

The theorems state the existence and uniqueness of `filter` as the unique fixpoint of `Filter`... except for the fact that instead of the expected equality we get bisimulation, coinductively defined as follows:

```
CoInductive bisim{T: Type}:Stream(T:=T) -> Stream(T:=T) -> Prop :=
|bisim_def: forall a s1 s2, bisim s1 s2 -> bisim(scons a s1)(scons a s2).
```

In the presentation of the first function-definition method from Section 3 we allowed ourselves, for simplicity of notation, to use equality instead of bisimulation. When translating informal mathematical reasoning to Coq such notation abuses and other similar approximations are revealed. Bisimulation is the natural equality between streams; by contrast, the standard equality of Coq is too strong. We note that, after having proved that bisimulation is a congruence relation, by importing a certain library (`Setoid`) one can perform rewriting with the fixpoint "equation" `filter_fix` in Coq.

Other examples

The companion Coq development also contains a definition of the stream of Fibonacci numbers, which, like the filter function is not accepted by Coq's builtin coinduction mechanisms. There is also a construction for a CPO of *colists*, which can be seen as the union of finite lists and streams. Accordingly, colists have a constructor `nil` for the empty colist, in addition to `bot` and `scons` like in the above definition of streams. The filter function on colists, defined as the unique fixpoint of a certain functional, turns out to be quite different from the filter function of streams: it is total on the subtype of "well-formed" colists (those that do not contain `bot`) and uses a non-executable "oracle" to determine whether its current argument is such that none of its elements satisfy the filtering predicate. If this is the case, the function returns `nil`, otherwise, it behaves like the filter function for streams.

5.3 Second method

Unlike the first method, in which each individual coinductive type has to be organized as a CPO, the second method provides a generic construction of CPOs, if some assumptions are met. Particular CPOs can be defined as instances of the generic notions, by providing definitions and lemmas that instantiate the assumptions. Corecursive functions between CPOs can then be defined.

5.3.1 Generic CPO

The generic construction of CPOs requires a set `Cc` (C°, in Section 4.1), a least element, and an order relation `ordc` (for $\preceq^\circ$), which must be weakly total. There is also a measure `mu` (for μ) compatible with the strict order. These requirements are gathered in a Coq *module type*:

```
Parameter Cc: Type.
Parameter bot: Cc.
Parameter ordc: Cc-> Cc-> Prop.
Parameter bot_is_least: forall x,ordc bot x.
Parameter ordc_refl: forall x,ordc x x.
Parameter ordc_trans: forall x y z,ordc x y->ordc y z->ordc x z.
Parameter ordc_antisym: forall x y,ordc x y->ordc  y x->x=y.
Parameter ordc_wtot: forall x y z,ordc x z->ordc y z->ordc x y\/ordc y x.
Parameter mu: Cc -> nat.
Parameter mu_sordc: forall x y,ordc x y-> x<>y-> mu x<mu y.
```

The type `Cc` is "extended" to a type `C` by adding equivalence classes of ascending sequences of elements in `Cc`, and the order `ordc` is extended to a relation `ord`, which is then proved to be an order:

```
Inductive C:Type :=
|elt:forall (e:Cc),C
|cls:forall (ec:EqClass),C.
Inductive ord : C-> C-> Prop :=
|elt_elt: forall  e1 e2,ordc e1 e2 -> ord (elt e1)(elt e2)
|elt_cls: forall e ec,
    (forall t, in t ec-> exists n, ordc e (nth t n) )->
    ord (elt e)(cls ec)
|cls_cls: forall ec,ord (cls ec)(cls ec).
```

More information about our encoding of equivalence classes is given at the end of this section. The type `C` is obtained by "wrapping" elements in `Cc` into a constructor `elt` and equivalence classes into a constructor `cls`. The relation `ord` has three cases, corresponding the three cases by which $\preceq^{\circ}$ is extended to $\preceq$ in Definition 22. Then, the limit of an increasing sequence of elements in `C` is defined:

```
Definition lim (s:Seq(T:=C))(H:increasing ord s):C :=
  match (excluded_middle_informative (stabilizing s)) with
  | left stab =>
    let (c, _) := constructive_indefinite_description _ stab in c
    |right nostab =>
     (cls (class_of (exist _ (fun n => extract_elt (s n))
                      (conj (extract_elt_incr _ Hinc nostab)
                              (incr_nostab_nostab _ Hinc nostab)))))end.
```

Like in Definition 25, the code for `lim` needs to decide whether its argument `s` is stabilizing or not. This is not decidable, because a decision procedure would have to examine a whole infinite sequence. We make it decidable by proving a theorem called `excluded_middle_informative` stating that every proposition is decidable: `forall P,{P}+{~P}` – a consequence of classical logic and `constructive_indefinite_description`. Applying that theorem to `(stabilizing s)` leads to two cases: if the sequence is stabilizing (with `stab` being a proof of stabilization) then the value to which it stabilizes is "fetched" by `constructive_indefinite_description`. If the sequence is not stabilizing (with `nostab` being a proof of non-stabilization) then, intuitively the equivalence class of `s` should be returned – except for the fact that `s` is a sequence over `C` and we only have equivalence classes of ascending sequences over `Cc`. Various wrappers, conversion operations, and proof terms are used to produce the adequate equivalence class. Of course, none of these details were visible in the mathematical definition of the limit (Definition 25), but in Coq all the details are exposed. The proof of the fact that `lim` actually computes the least upper bound of its argument amounts to a similar exposure and management of many details, none of which is visible in the mathematical statements – Theorem 26 and its proof.

On equivalence classes

There is no universally accepted way for expressing equivalence classes modulo a given equivalence relation in Coq. One option, supported by the tool's standard library, is to use *setoids*, which are a triple consisting of a type, a binary relation on the type, and a proof that the relation is an equivalence. This approach is mainly used to obtain a generalized rewriting, using the setoid's equivalence relation (which moreover needs to be proved to be a congruence for the contexts under which rewriting is desired) instead of equality. For example, rewriting using bisimulation of streams falls in this category. However, in the present context, we just need equivalence classes for their own sake. Rewriting is not an issue, and using the powerful but complicated machinery of setoids did not seem cost-effective. We therefore opted for a more direct approach that uses axioms from the

standard library: `constructive_indefinite_description` for obtaining a representative of a class; functional extensionality (two functions are equal iff they are pointwise equal) and propositional extensionality (propositional equality coincides with equivalence) for proving that if two elements are in the equivalence relation they are in the same equivalence class. In standard mathematics these properties are implicitly assumed, but in Coq they have to be explicitly assumed since they are not provable otherwise.

5.3.2 The CPO of finite and Rose trees

In order to obtain this CPO the parameters of the generic CPO (the type `Cc`, the relation `ordc`, the function `mu`, and the various constraints relating them) have to be instantiated with actual definitions and lemmas. This essentially amounts to encoding the content of Example 29 in Coq. The hardest part was establishing that the relation `ordc` is transitive; several nontrivial lemmas about cutting trees at given heights had to be proved. Perhaps the most difficult part of all the development effort was to convince ourselves that weak totality of the order is a crucial requirement, and therefore to abandon the apparently natural "prefix order", also defined in Example 29, which does not have this property.

5.3.3 The mirror function

Defining a function using the second method is composed of a generic part, which assumes two generic CPOs and a function between their "finite parts" that has to satisfy a productiveness constraint (Definition 30). Accordingly, in Coq we write a module type where such a function and its productiveness requirement are assumed. Any module that implements that module type gains access to the corecursive function definition method described at the end of Section 4.2. A recursive `fmirror` function between finite trees is written in such a module, and by the generic mechanism described above, this function is transformed into a corecursive function `mirror` between Rose trees.

Finally, to gain confidence in the obtained definition we define functions `fpos` and `fpos_rev` (cf. Example 33) that compute labels at given positions in finite trees; transform these functions into `pos` and `pos_rev` that do the corresponding operations on Rose trees; and prove the following lemma:

```
Lemma mirror_pos: forall p t, pos p (mirror t) = pos_rev p t.
```

6 Conclusion, related work, and future work

This paper presents two methods for defining corecursive functions that go beyond the guarded-by-constructor setting available in the Coq proof assistant. The first method reuses the dedicated coinduction mechanisms available in Coq, which works as long as the underlying coinductive datatypes are not mutually dependent with inductive types. The second method is not subject to this restriction, as it does not rely on Coq's coinduction mechanisms but redefines them, at the cost of some additional work. Both methods have in common the interpretation of maximal values in CPOs as well-defined corecursive values, and they both rely on a mathematical notion of *productiveness* that captures the corresponding intuitive notion of productiveness (the ability of a function to eventually generate, for each input, an arbitrarily close approximation of the corresponding output). Both methods are implemented in Coq and are illustrated by defining corecursive functions that Coq's dedicated mechanisms reject. This gain in expressiveness is obtained at the cost of using axioms from

the standard library of Coq, which are known not to introduce inconsistencies: using them amounts to losing constructiveness, but gaining access to standard mathematical reasoning. Both methods were presented independently of Coq; especially the second one, which *is* independent from Coq's builtin mechanisms for corecursion, could be implemented in other proof assistants. An interesting target is Lean [14], a dependently-typed language and proof assistant that includes the additional feature of *quotient types* that would naturally encode equivalence classes in the second method.

The methods we propose transform a problem currently without solution (defining corecursive functions that do not satisfy the guardedness condition) into a problem that is solvable: defining and reasoning about functions that approximate the function under definition. In practice the approximating functions are recursive, as can be seen from the examples in the paper (Examples 14 and 31) and from the additional ones in the companion Coq development. Now, if for a given corecursive function the corresponding approximating recursive functions are difficult to reason about, then applying our methods may be difficult. However, most of the difficulty does not arise from the methods, but from the intrinsic complexity of the corecursive function being defined.

Comparison with related work

We start with classical results and with their applications for the purpose of defining functions. Kleene's fixpoint theorem [19, Chapter 5] can be used to define functions as *least* fixpoints of *continuous* functionals over CPOs. A functional is continuous if it commutes with least upper bounds. The least fixpoint is the least upper bound of an increasing sequence of functions, obtained by iterating the functional starting from the constant "bottom" function. This has been formalized and used for defining recursive functions in Coq [5]. Unsurprisingly, they use the same kinds of axioms as we do.

In our first method we use the same iteration as in Kleene's fixpoint theorem to obtain a fixpoint, but require *productiveness* instead of continuity; and we obtain a unique fixpoint, not just a least fixpoint. The stronger fixpoint result, and the fact that productiveness is a natural requirement for corecursive functions, suggest that our method is well-adapted for the purpose of defining such functions.

Our second method has similarities with the classical construction of the real numbers based on equivalence classes of Cauchy sequences of rational numbers [10, Appendix A]. However, Cauchy sequences over a base set require the base set to be organized as a metric space, with a distance function satisfying certain properties. An approach for defining corecursive functions based on Cauchy sequences is mentioned in [13]. They use another classical result (Banach's fixpoint theorem [2]) to define corecursive functions as unique fixpoints of *eventually contracting* functionals. By contrast, we organize the base set as a CPO, use ascending sequences instead of Cauchy sequences, and (in the second method) do not use functionals, but a "finite version" of the corecursive function under definition, which has to satisfy a certain productiveness requirement to ensure a proper definition.

We now present related work about corecursion in proof assistants and similar formalisms. In Coq, corecursive function definitions have to satisfy a guardedness-by-constructors criterion. This criterion ensures a strong version of productiveness, namely, that each evaluation step produces a strictly closer approximation of the final result than the previous steps. By contrast, productiveness only requires that *eventually* a strictly closer approximation is obtained. In some cases, a function that is productive but unguarded can be transformed into an equivalent, guarded function. This has been done for the filter function on streams in [3] and generalized in [4] to other unguarded functions. Their idea is to use an ad-hoc

predicate stating that the definition under study is, in some sense, productive. However, their approach does not use a general, formal notion of productiveness, nor does it handle the case where corecursive calls are guarded by some non-constructor function, like the mirror function for Rose trees presented in this paper. Our approach is not subject to these limitations. In other related work, a constructive version of the CPO of streams in Coq is mentioned in [16] in the context of a coinductive formalization of Kahn networks. However, the author does not use her formalization of CPO to extend the class of corecursive stream functions admissible by Coq.

We note that *coinductive proofs* in Coq, which by default are subject to the same syntactical requirements as corecursive functions, can be performed using more relaxed, semantical requirements by using *parameterized coinduction* implemented in the *Paco* extension of Coq [12]. We have tried to adapt parameterized coinduction to corecursive function definition, but have given up because we found that it is not adaptable. Parameterized coinduction works for coinductive proofs, because, there, *witness terms* do not matter – any term of the right type will do. By contrast, in corecursive functions, witness terms do matter, since they expresses what the function is supposed to compute.

Agda [20] is also a dependently-typed programming language and proof assistant that offers support for corecursive function definition. In the core tool there is a guardedness checker similar to that of Coq, but more liberal as it allows, e.g., the definition of two mutually dependent functions, one of which is recursive and the other one, corecursive. This enables it to accept the definition of the mirror function on Rose trees, which Coq does not accept. Extensions to Agda with sized types [18] provide users with a uniform, automatic way of handling termination and productiveness, based on type annotations written by the user. The current implementation of sized types in Agda is unsound (cf. [20, chapter Safe Agda] and `https://github.com/agda/agda/issues/2820`).

Isabelle/HOL [21] is another major proof assistant which supports corecursive function definition. A guardedness criterion (there called *primitive corecursion*) similar to that of Coq and Agda is implemented [6], based on *bounded natural functors*, a conservative extension of Higher Order Logic. The framework has further been extended to accept function definitions that go beyond primitive corecursion [7]. Isabelle/HOL now accepts function definitions where corecursive calls can be guarded by functions other than constructors, provided the functions are proved to be *friendly* (essentially, a friendly function needs to destruct at most one constructor of input to produce one constructor of output). Unguarded corecursive calls, such as those in the filter function on streams, are also accepted, provided they are proved to eventually produce a constructor of output. Like in our approach, all proof obligations are the responsibility of the user. They have the additional advantage of using no supplementary axioms, as those of Higher Order Logic are expressive enough.

Beyond generic proof assistants, support for corecursion also exists in tools targeting particular languages. For example, Dafny is a specification and verification language dedicated to the C# language, which has support for corecursive function definition [15], based on a guardedness criterion similar to those existing in the already mentioned tools. Coinductive proofs are also supported.

Finally, beyond the area of formal verification, it is very worth mentioning the Haskell functional language, which offers support for corecursive function definition by means of lazy evaluation.

Future work

We have encountered corecursive functions that are productive yet do not obey the guarded-by-constructors criterion in our planned future work. The Prelude dataflow synchronous programming language [9] has a flow sampling construction whose semantics is best described using a filter function on colists (which we have defined in the companion Coq development as an instance of our first method). This opens the way to a mechanized semantics of Prelude in Coq, which would then enable program verification and semantically correct code generation for the language. While formalizing in Coq the paper [17] about the semantics of dataflow languages we have encountered unguarded corecursive functions on streams that can also be defined using our first method. More speculative future work includes a comparison and possible cross-fertilization of our approach with the sized-type approach of Agda and the bounded-natural-functor approach of Isabelle/HOL.

References

1 *The Coq Proof Assistant.* URL: `https://coq.inria.fr/`.

2 S. Banach. Sur les opérations dans les ensembles abstraits et leur applications aux équations intégrales. *Fundam. Math.*, 3:133–181, 1922.

3 Y. Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115, 2005.

4 Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in Coq. In *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008, Budapest, Hungary, April 4-6, 2008*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 25–47, 2008.

5 Y. Bertot and V. Komendantsky. Fixed point semantics and partial recursion in Coq. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 89–96, 2008.

6 J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL.* URL: `https://isabelle.in.tum.de/doc/datatypes.pdf`.

7 J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. *Defining Nonprimitively (Co)recursive Functions in Isabelle/HOL.* URL: `https://isabelle.in.tum.de/dist/Isabelle2021/doc/corec.pdf`.

8 A. Chlipala. *Certified Programming with Dependent Types.* MIT Press, 2013.

9 J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints.* PhD thesis, Institut Supérieur de l'Aéronautique et de l'Espace, Toulouse, France, 2009.

10 S. R. Ghorpade and B. V. Limaye. *A Course in Calculus and Real Analysis.* Undergraduate Texts in Mathematics. Springer, 2018.

11 E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

12 C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013.

13 D. Kozen and N. Ruozzi. Applications of metric coinduction. *Log. Methods Comput. Sci.*, 5(3), 2009.

14 *The Lean Proof Assistant.* URL: `https://leanprover.github.io/`.

15 K. Rustan M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2014.

16 C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. Available at `https://www.lri.fr/~paulin/PUBLIS/paulin07kahn.pdf`.

17 T. Uustalu and V. Vene. The essence of dataflow programming. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.

18 N. Veltri and N. van der Weide. Guarded recursion in agda via sized types. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

19 G. Winskel. *The Formal Semantics of Programming Languages, an introduction.* MIT Press, 1993.

20 *The Agda Proof Assistant.* `https://agda.readthedocs.io/en/v2.6.2/getting-started/what-is-agda.html`.

21 *The Isabelle/HOL Proof Assistant.* URL: `https://isabelle.in.tum.de/`.

REST: Integrating Term Rewriting with Program Verification

Zachary Grannan
University of British Columbia, Vancouver, Canada

Niki Vazou
IMDEA Software Institute, Madrid, Spain

Eva Darulova[1]
Uppsala University, Sweden

Alexander J. Summers
University of British Columbia, Vancouver, Canada

Abstract

We introduce REST, a novel term rewriting technique for theorem proving that uses online termination checking and can be integrated with existing program verifiers. REST enables flexible but terminating term rewriting for theorem proving by: (1) exploiting newly-introduced term orderings that are more permissive than standard rewrite simplification orderings; (2) dynamically and iteratively selecting orderings based on the path of rewrites taken so far; and (3) integrating external oracles that allow steps that cannot be justified with rewrite rules. Our REST approach is designed around an easily implementable core algorithm, parameterizable by choices of term orderings and their implementations; in this way our approach can be easily integrated into existing tools. We implemented REST as a Haskell library and incorporated it into Liquid Haskell's evaluation strategy, extending Liquid Haskell with rewriting rules. We evaluated our REST implementation by comparing it against both existing rewriting techniques and E-matching (as used in most SMT solvers) and by showing that it can be used to supplant manual lemma application in many existing Liquid Haskell proofs.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases term rewriting, program verification, theorem proving

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.13

Related Version *Extended Version*: `https://arxiv.org/abs/2202.05872` [26]

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.12`

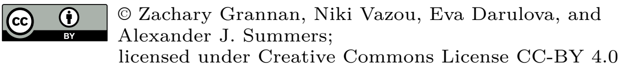
Funding This work was supported by the Juan de la Cierva grants IJC2019-041599-I, the HaCrypt ONR project N00014-19-1-2292, and the ERC starting grant CRETE (101039196). We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

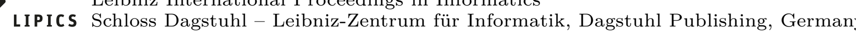
Acknowledgements We thank Jonathan Chan, Eric Conlon, Rui Ge, Paulette Koronkevich and the anonymous reviewers for their helpful and constructive feedback.

1 Introduction

For all disjoint sets s_0 and s_1, the identity $(s_0 \cup s_1) \cap s_0 = s_0$ can be proven in many ways. Informally accepting this property is easy, but a machine-checked formal proof may require the instantiation of multiple set theoretic axioms. Analogously, further proofs relying on this

[*] This work was partly done while the author was at MPI-SWS.


© Zachary Grannan, Niki Vazou, Eva Darulova, and Alexander J. Summers;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 13; pp. 13:1–13:29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

identity may themselves need to apply it as a previously-proven lemma. For example, proving functional correctness of any program that relies on a set data structure typically requires the instantiation of set-related lemmas. Manual instantiation of such universally quantified equalities is tedious, and the burden becomes substantial for more complex proofs: a proof author needs to identify exactly which equalities to instantiate and with which arguments; in the context of program verification, a wide variety of such lemmas are typically available. Given this need, most program verifiers provide some automated technique or heuristics for instantiating universally quantified equalities.

For the wide range of practical program verifiers that are built upon SMT solvers (e.g., [35, 23, 51, 39, 47, 45]), quantified equalities can naturally be expressed in the SMT solver's logic. However, relying solely on such solvers' E-matching techniques [19] for quantifier instantiation (as the majority of these verifiers do) can lead to both non-termination and incompletenesses that may be unpredictable [34] and challenging to diagnose [7]. Theory and techniques for proving termination and completeness of encodings using E-matching for equality reasoning is relatively unexplored [21]; the inherent treatment of terms modulo equalities makes standard term orderings based on term structure unsound.

A classical alternative approach to automating equality reasoning is *term rewriting* [28], which can be used to encode lemma properties as (directed) rewrite rules, matching terms against the existing set of rules to identify potential rewrites; the termination of these systems is a well-studied problem [16]. Although SMT solvers often perform rewriting as an internal simplification step, verifiers built on top typically cannot access or customize these rules, e.g., to add previously-proved lemmas as rewrite rules. By contrast, many mainstream proof assistants (e.g., `Coq` [11], `Isabelle/HOL` [40], `Lean` [5]) provide automated, customizable term rewriting tactics. However, the rewriting functionalities of mainstream proof assistants either do not ensure the termination of rewriting (potentially resulting in divergence, for example Isabelle) or enforce termination checks that are overly restrictive in general, potentially rejecting necessary rewrite steps (for example, Lean).

In this paper, we present *`REST` (REwriting and Selecting Termination orderings)*: a novel technique that equips program verifiers with automatic lemma application facilities via term rewriting, enabling equational reasoning with complementary strengths to E-matching-based techniques. While term rewriting in general does not guarantee termination, our technique weaves together three key technical ingredients to automatically generate and explore guaranteed-terminating restrictions of a given rewriting system while typically retaining the rewrites needed in practice: (1) `REST` compares terms using well-quasi-orderings derived from (strict) simplification orderings; thereby facilitating common and important rules such as commutativity and associativity properties. (2) `REST` simultaneously considers an entire family of term orderings; selecting the appropriate term ordering to justify rewrite steps *during term rewriting itself.* (3) `REST` allows integration of an *external oracle* that generates additional steps outside of the term rewriting system. This allows the incorporation of reasoning steps awkward or impossible to justify via rewriting rules, all without compromising the termination and relative completeness guarantees of our overall technique.

Contributions and Overview. We make the following contributions:

1. We design and present a new approach (`REST`) for applying term rewriting rules and simultaneously selecting appropriate term orderings to permit as many rewriting steps as possible while guaranteeing termination (Sec. 3).
2. We introduce ordering constraint algebras, an abstraction for reasoning effectively about multiple (and possibly infinitely many) term orderings simultaneously (Sec. 4).

Name	*Formula*
idem-union	$X \cup X \; = \; X$
idem-inter	$X \cap X \; = \; X$
empty-union	$X \cup \emptyset \; = \; X$
empty-inter	$X \cap \emptyset \; = \; \emptyset$
commut-union	$X \cup Y \; = \; Y \cup X$
symm-inter	$X \cap Y \; = \; Y \cap X$
distrib-union	$(X \cup Y) \cap Z \; = \; (X \cap Z) \cup (Y \cap Z)$
distrib-inter	$(X \cap Y) \cup Z \; = \; (X \cup Z) \cap (Y \cup Z)$
assoc-union	$X \cup (Y \cup Z) \; = \; (X \cup Y) \cup Z$

Figure 1 Set identities used for examples in this section. Variables X, Y, Z are implicitly quantified. We write the binary functions $\cup, \cap$ infix; along with (nullary) $\emptyset$ these are fixed function symbols.

3. We introduce and formalize recursive path quasi-orderings (RPQOs) derived from the well-known recursive path ordering [15] (Sec. 4.1.2). RPQOs are more permissive than classical RPOs, and so let us prove more properties.
4. We formalize and prove key results for our technique: soundness, relative completeness, and termination (Sec. 5).
5. We implement `REST` as a stand-alone library, and integrate the `REST` library into Liquid Haskell to facilitate automatic lemma instantiation (Sec. 6).
6. We evaluate `REST` by comparison to other term rewriting tactics and E-matching-based axiomatization, and show that `REST` can simplify equational reasoning proofs (Sec. 7).

We discuss related work in Sec. 8; we begin (Sec. 2) by identifying five key challenges of a reliable and automatic integration of term rewriting into a program verification tool.

2 Five Challenges for Automating Term Rewriting

In this section, we describe *five key challenges* that naturally arise when term rewriting is used for program verification and outline how `REST` is designed to address them. To illustrate the challenges, we use simple verification goals that involve uninterpreted functions and the set operators ($\emptyset$, $\cup$, $\cap$) that satisfy the standard properties of Figure 1. The variables x, y, z are implicitly quantified[2] in these rules. In formalizations of set theory, such properties may be assumed as (quantified) axioms, or proven as lemmas and then used in future proofs.

Term rewriting systems (defined formally in Sec. 5) are a standard approach for formally expressing and applying equational reasoning (rewriting terms via known identities). A term rewriting system consists of a finite set of *rewrite rules*, each consisting of a pair of a *source term* and a *target term*, representing that terms matching a rule's source can be replaced by corresponding terms matching its target. For example, the rewrite rule $X \cup \emptyset \to X$ can replace set unions of some set X and the empty set with the corresponding set X. Rewrite rules are applied to a term t by identifying some subterm of t which is equal to a rule's source under some substitution of the source's free variables (here, X, but not constants such as $\emptyset$); the subterm is then replaced with the correspondingly substituted target term. This rewriting step *induces an equality* between the original and new terms. For instance, the example rewrite rule above can be used to rewrite a term $f(s_0 \cup \emptyset)$ into $f(s_0)$, inducing an equality between the two.

[2] over sets; we omit explicit types in such formulas, whose type-checking is standard.

Rewrite rules classically come with two restrictions: the free variables of the target must all occur in the source and the source must not be a single variable. This precludes rewrite rules which invent terms, such as $\emptyset \to X \cap \emptyset$, and those that trivially lead to infinite derivations. Under these restrictions, the first four identities induce rewrite rules from left-to-right (which we denote by e.g., *idem-inter*$\to$), while the remaining induce rewrite rules in both directions (e.g., *assoc-union*$\to$ vs. *assoc-union*$\leftarrow$).

Next, we present a simple proof obligation taken from [36] in the style of equational reasoning (*calculational proofs*) supported in the Dafny program verifier [35].

▶ **Example 1.** We aim to prove, for two sets s_0 and s_1 and some unary function f on sets, that, if the sets are disjoint (that is, $s_1 \cap s_0 = \emptyset$), then $f((s_0 \cup s_1) \cap s_0) = f(s_0)$.

Equational Proof:

$$\begin{array}{rcll} f((s_0 \cup s_1) \cap s_0) & = & f((s_0 \cap s_0) \cup (s_1 \cap s_0)) & \textit{(distrib-union}\to\textit{)} \\ & = & f(s_0 \cup (s_1 \cap s_0)) & \textit{(idem-inter}\to\textit{)} \\ & = & f(s_0 \cup \emptyset) & \textit{(disjointness ass.}\to\textit{)} \\ & = & f(s_0) & \textit{(empty-union}\to\textit{)} \end{array}$$

(Possible Term Ordering, as explained shortly: RPO instance with $\cap > \cup$)

This manual proof closely follows the user annotations employed in the corresponding Dafny proof [36]; the application of the function f serves only to illustrate equational reasoning on subterms. Every step of the proof could be explained by term rewriting, hinting at the possibility of an *automated* proof in which term rewriting is used to solve such proof obligations. In particular, taking the term rewriting system naturally induced by the set identities of Figure 1 *along with* the assumed equality expressing disjointness of s_0 and s_1 results in a term rewriting system in which the four proof steps are all valid rewriting steps.

In the remainder of the section, we consider what it would take to make term rewriting effective for reliably automating such verification tasks. Perhaps unsurprisingly, there are multiple problems with the simplistic approach outlined so far. The first and most serious is that term rewriting systems in general *do not guarantee termination*; a proof search may continue indefinitely by repeatedly applying rewrite rules. For example, the rules *distrib-union* and *distrib-inter* can lead to an infinite derivation $(s_0 \cup s_1) \cap s_2 \to (s_0 \cap s_2) \cup (s_1 \cap s_2) \to (s_0 \cup (s_1 \cap s_2)) \cap (s_2 \cup (s_1 \cap s_2)) \to \ldots$

Challenge 1: Unrestricted term rewriting systems do not guarantee termination.

To ensure termination (as proved in Theorem 22) `REST` follows the classical approach of restricting a term-rewriting system to a variant in which sequences of term rewrites (*rewrite paths*) are allowed only if each consecutive pair of terms is *ordered* according to some term ordering which rules out infinite paths.

For example, *Recursive path orderings* (RPOs) [15] define well-founded orders $>_{\mathcal{T}}$ on terms $\mathcal{T}$ based on an underlying well-founded strict partial order $>$ on *function symbols*. Intuitively, such orderings use $>$ to order terms with different top-level function symbols, combined with the properties of a *simplification order* [14] (e.g., compatibility with the subterm relation). Different choices of the underlying $>$ parameter yield different RPO instances that order different pairs of terms; in particular, potentially allowing or disallowing certain rewrite paths.

In Example 1, an RPO based on a partial order where $\cap > \cup$ and $\cap > \emptyset$ permits all the rewriting steps, that is, the left-hand-side of each equation is greater than the right-hand-side.

Sadly, this ordering will not permit the rewriting steps required by our next example.

▶ **Example 2.** We aim to prove, for two sets s_0 and s_1 and some unary function f on sets, that, if s_1 is a subset of s_0 (that is, $s_0 \cup s_1 = s_0$), then $f((s_0 \cap s_1) \cup s_0) = f(s_0)$.

$$
\begin{array}{llll}
\textit{Equational Proof:} \quad f((s_0 \cap s_1) \cup s_0) & = & f((s_0 \cup s_0) \cap (s_1 \cup s_0)) & \textit{(distrib-inter}\rightarrow\textit{)} \\
& = & f(s_0 \cap (s_1 \cup s_0)) & \textit{(idem-union}\rightarrow\textit{)} \\
& = & f(s_0 \cap (s_0 \cup s_1)) & \textit{(commut-union}\rightarrow\textit{)} \\
& = & f(s_0 \cap s_0) & \textit{(subset ass.}\rightarrow\textit{)} \\
& = & f(s_0) & \textit{(idem-inter}\rightarrow\textit{)}
\end{array}
$$

(Possible Term Ordering: RPQO instance, explained shortly, with $\cup > \cap$)

An RPO based on an ordering where $\cap > \cup$ (as required by Example 1) will not permit the first step of this proof (since the RPO ordering first compares the top level function symbols). Instead, this step requires an RPO based on an ordering where $\cup > \cap$. To accept *both* this proof step *and* the Example 1 we need different restrictions of the rewrite rules for different proofs; in particular, different rewrite paths may be ordered according to RPOs that are based on different function orderings.

To generalize this problem we will call RPOs a term ordering *family* that is *parametric* with respect to the underlying function ordering. Thus, a concrete RPO term ordering (called an *instance* of the family) is obtained after the parametric function ordering is instantiated. With this terminology, the next challenge can be stated as follows:

Challenge 2: Different proofs require different term orderings within a family.

Note that enumerating all term orderings in a term ordering family is typically impractical (this set is often very large and may be infinite). To address this challenge, `REST` uses a novel algebraic structure (Sec. 4.2) to allow for an abstract representation of sets of term orderings with which one can efficiently check whether any instance of a chosen term ordering family can orient the necessary rewrite steps to complete a proof.

Going back to Example 2, the RPO instance with $\cup > \cap$ will permit all the steps, apart from the commutativity axiom expressed by *(commut-union→)*. To permit this step we need an ordering for which $t_1 \cup t_2 >_{\mathcal{T}} t_2 \cup t_1$. But for RPO instances, as well as for many other term orderings, the terms $t_1 \cup t_2$ and $t_2 \cup t_1$ are equivalent and thus cannot be oriented; associativity axioms are also similarly challenging. Since many proofs require such properties, it is important in practice for rewriting to support them.

Challenge 3: Strict orderings restrict commutativity and associativity steps.

To address this challenge `REST` relaxes the strictness constraint by requiring the chosen term ordering family to consist (only) of *thin well-quasi-orderings* (Def. 5). Intuitively, such orderings permit rewriting to terms which are *equal* according to the ordering, but such equivalence classes of terms must be finite. In Sec. 4 we show how to lift well-known families of term orderings to more-permissive families of thin well-quasi-orders. In particular, we show how to lift RPOs to a particularly powerful family of term orderings that we call *recursive path quasi-orderings (RPQOs)* (Def. 10), whose instances allow us to accept Example 2.

Despite the permissiveness of RPQOs, there remain some rewrite derivations that will be rejected by all term orderings in the RPQO family. For example, consider the following proof that set union is monotonic with respect to the subset relation:

▶ **Example 3.** We aim to prove, for sets s_0, s_1, and s_2, that, if s_1 is a subset of s_0 (that is, $s_0 \cup s_1 = s_0$), then $(s_2 \cup s_1) \cup (s_2 \cup s_0) = s_2 \cup s_0$.

Equational Proof:

$$
\begin{aligned}
(s_2 \cup s_1) \cup (s_2 \cup s_0) &= s_2 \cup (s_1 \cup (s_2 \cup s_0)) && \textit{(assoc-union}\leftarrow\textit{)}\\
&= s_2 \cup ((s_1 \cup s_2) \cup s_0) && \textit{(assoc-union}\rightarrow\textit{)}\\
&= s_2 \cup ((s_2 \cup s_1) \cup s_0) && \textit{(commut-union}\rightarrow\textit{)}\\
&= s_2 \cup (s_2 \cup (s_1 \cup s_0)) && \textit{(assoc-union}\leftarrow\textit{)}\\
&= s_2 \cup (s_2 \cup (s_0 \cup s_1)) && \textit{(commut-union}\rightarrow\textit{)}\\
&= s_2 \cup (s_2 \cup s_0) && \textit{(subset ass.}\rightarrow\textit{)}\\
&= (s_2 \cup s_2) \cup s_0 && \textit{(assoc-union}\rightarrow\textit{)}\\
&= s_2 \cup s_0 && \textit{(idem-union}\rightarrow\textit{)}
\end{aligned}
$$

(Possible Term Ordering: any KBQO instance)

The above rewrite rule steps cannot be oriented by any RPQO, but are trivially oriented by a quasi-ordering that is based on the syntactic size of the term, e.g., a quasi-ordering based on the well-known Knuth-Bendix family of term orderings [31]. Yet, a Knuth-Bendix quasi-ordering (KBQO, defined in Sec. 4) cannot be used on our previous two examples; fixing even a single choice of term ordering *family* would still be too restrictive in general.

Challenge 4: Some proofs require different families of term orderings.

To address this challenge, REST (Sec. 3.2) is defined parametrically in the choice and representation of a term ordering family.

Finally, although equational reasoning is powerful enough for these examples, general verification problems usually require reasoning beyond the scope of simple rewriting. For example, simply altering Example 1 to express the disjointness hypothesis instead via cardinality as $|s_0 \cap s_1| = 0$ means that, to achieve a similar proof, reasoning within the theory of sets is necessary to deduce that this hypothesis implies the equality needed for the proof; this is beyond the abilities of term rewriting.

Challenge 5: Program verification needs proof steps not expressible by rewriting.

To address this challenge, our REST approach allows the integration of an external oracle that can generate equalities not justifiable by term rewriting (Sec. 3.3).

3 The REST Approach

We develop REST to tackle the above five challenges and integrate a flexible, expressive, and guaranteed-terminating term rewriting system with a verification tool. REST consists of an interface that defines term orderings and an algorithm that explores the terminating rewrite paths. In Sec. 3.1 we describe the representation of term orderings in REST and how they address Challenges 2 and 4. In Sec. 3.2 we describe the REST algorithm that is parametric to these orderings and Sec. 3.3 describes the integration with external oracles (Challenge 5).

3.1 Representation of Term Orderings in REST

Rather than considering individual term orderings, REST operates on indexed sets (families) of term orderings (whose instances must all be thin well-quasi-orderings [Def. 5]).

▶ **Definition 4** (Term Ordering Family). *A* term ordering family Γ *is a set of thin well-quasi-orderings on terms, indexed by some parameters P. An* instance *of the family is a term ordering obtained by a particular instantiation of P.*

For example, the recursive path ordering is defined parametrically with respect to a precedence on function symbols, and therefore defines a term ordering family indexed by this choice of function symbol ordering.

A core concern of `REST` is determining whether any instance of a given term ordering family can orient a rewrite path. However, term ordering families cannot directly compare terms; doing so requires choosing an ordering inside the family. The root of Challenge 2 is that choosing an ordering in advance is too restrictive: different orderings are necessary to complete different proofs. The idea behind `REST`'s search algorithm is to address this challenge by simultaneously considering all orderings in the family when considering rewrite paths and continuing the path so long as it can be oriented by *any* ordering.

To demonstrate the technique, we show how `REST`'s approach can be derived from a naïve algorithm. The purpose of the algorithm is to determine if any ordering in a family Γ can orient a path $t_1 \to \ldots \to t_n$; i.e., if there is a $>_{\mathcal{T}} \in \Gamma$ such that $t_1 >_{\mathcal{T}} \ldots >_{\mathcal{T}} t_n$.

`orients` : (Set $O \times$ List $\mathcal{T}$) $\to$ *Bool*		`orients` : ($OCA \times$ List $\mathcal{T}$)) $\to$ *Bool*
`orients`$(\Gamma, ts) =$		`orients`$(\langle \top, \mathit{refine}, \mathit{sat}\rangle, ts) =$
$os := \Gamma$;	(1)	$c := \top$;
for $i \in 1$ **to** $\lvert ts \rvert - 1$ {		**for** $i \in 1$ **to** $\lvert ts \rvert - 1$ {
$os := \{>_{\mathcal{T}} \in os \mid ts_i >_{\mathcal{T}} ts_{i+1}\}$;	(2)	$c := \mathit{refine}(c, ts_i, ts_{i+1})$;
if $(os = \emptyset)$	(3)	**if** (**not**$(\mathit{sat}(c))$)
return *false*;		**return** *false*;
}		}
return *true*;		**return** *true*;

Figure 2 Two algorithms that determine if an ordering in the term ordering family Γ can orient a path of terms ts. **Left** presents the naïve, exhaustive algorithm. **Right** is using the ordering constraint algebra $\langle \top, \mathit{refine}, \mathit{sat}\rangle$ that returns true iff an ordering in Γ can orient ts without explicitly constructing any term orderings. $O$is the type of a term ordering.

The naïve algorithm is depicted on the left of Figure 2. The naïve algorithm works iteratively, computing the set of orderings os that can orient an increasingly-long path, short-circuiting if the set becomes empty. The algorithm enumerates each ordering in Γ and compares terms with each ordering (potentially multiple times). Unfortunately, this enumeration is not practical: some term ordering families have infinite or prohibitively large numbers of instances. `REST` avoids these issues by allowing the set of term orderings to be abstracted via a structure called an Ordering Constraint Algebra (OCA, Def. 14 of Sec. 4.2).

An OCA for a term ordering family Γ consists of a type C along with four parameters $\gamma : C \to \mathcal{P}(\Gamma)$, $\top : C$, $\mathit{refine} : C \to \mathcal{T} \to \mathcal{T} \to C$, and $\mathit{sat} : C \to \mathit{Bool}$. C is a type whose elements *represent* subsets of Γ. The function γ is the *concretisation function* of the OCA, not needed programmatically but instead defining the *meaning* of elements of C in terms of the subsets of the term ordering family they represent. The remaining three functions correspond to the operations on sets of term orderings used in lines (1), (2), and (3) of the naïve algorithm. $\top$ represents the set of all term orderings in Γ, $\mathit{refine}(c, t, u)$ filters the set of orderings represented by c to include only those where $t >_{\mathcal{T}} u$, and $\mathit{sat}(c)$ is a predicate that returns true if the set of orderings represented by c is nonempty. Figure 2 on the right shows

```
REST : (OCA × R × T × (T → Set T)) → Set T
REST(⟨⊤, refine, sat⟩, R, t0, E) =
  o := ∅;
  p := [([t0], ⊤)];
  while (p is not empty){
    pop(ts, c) from p;
    t := last ts;
    o := o ∪ {t};
    foreach (t' such that t' ∉ ts ∧ (t →R t' ∨ t' ∈ E(t))){
      if (t' ∈ E(t) ∨ (t →R t' ∧ sat(refine(c, t, t')))){
        push (ts ++ [t'], refine(c, t, t')) to p
      }
    }
  }
  return o;
```

Figure 3 The REST algorithm.

how the ordering constraint algebra can be used to perform an equivalent computation to the naïve algorithm, without explicitly instantiating sets of term orderings. The OCA plays a role similar to abstract interpretation in a program analysis, where C is an abstraction over sets of term orderings, and the results of the abstract operations on C correspond to their concrete equivalents. Namely, we have $\gamma(\top) = \Gamma$, $\gamma(\mathit{refine}(c, t_l, t_r)) = \{\succcurlyeq \mid \succcurlyeq \in \gamma(c) \wedge t_l \succcurlyeq t_r\}$, and $\mathit{sat}(c) \Leftrightarrow \gamma(c) \neq \emptyset$.

The ordering constraint algebra enables three main advantages compared to direct computation with sets of term orderings:

1. The number of term orderings can be very large, or even infinite, thus making enumeration of the entire set intractable.
2. An OCA can provide efficient implementations for *refine* and *sat* by exploiting properties of the term ordering family. Comparing terms using the constituent term orderings requires repeating the comparison for each ordering, despite the fact that most orderings will differ in ways that are irrelevant for the comparison.
3. The OCA does not impose any requirements on the type of C or the implementation of $\top$, *refine*, and *sat*. For example, an OCA can use $\top$ and *refine* to construct logical formulas, with *sat* using an external solver to check their satisfiability. Alternatively, it could define C to be sets of term orderings that are reasoned about explicitly, and implement $\top$, *refine*, and *sat* as the operations of the naïve algorithm.

We now describe how the REST algorithm uses the OCA to explore rewrite paths.

3.2 The REST Algorithm

Figure 3 presents the REST algorithm. The algorithm takes four parameters. The first parameter is an OCA $\langle \top, \mathit{refine}, \mathit{sat} \rangle$, as discussed above. The algorithm's second parameter, R, is a finite set of term rewriting rules (not required to be terminating); for example, we could pass the oriented rewrite rules corresponding to Figure 1. The third parameter t_0 is the term from which term rewrites are sought. The final parameter $\mathcal{E}$ acts as an external oracle, generating additional rewrite steps that need *not* follow from the term rewriting rules R. To simplify the explanation, we will initially assume that $\mathcal{E} = \lambda t.\emptyset$, i.e., this parameter

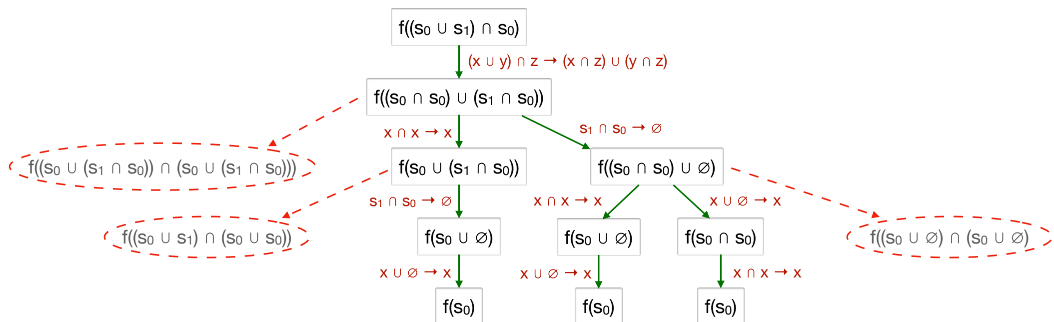


Figure 4 A visualization of REST running on the term from Example 1. Each path through the tree shown represents a rewrite path uncovered by our algorithm; the edge labels show the rewrite rule applied. The red dotted lines indicate rewrite steps rejected by REST.

has no effect. Our algorithm produces a set of terms, each of which are reachable by *some* rewrite path beginning from t_0, and for which *some* ordering allows the rewrite path. The algorithm addresses Challenge 1 (termination; Theorem 22) because every path must be finite: no ordering could orient an infinite path.

REST operates in worklist fashion, storing in p a list of pairs (ts, c) where ts is a non-empty list of terms representing a rewrite path already explored (the head of which is always t_0) and c tracks the ordering constraints of the path so far. The set o records the output terms (initially empty): all terms discovered equal to t_0 via the rewriting paths explored.

While there are still rewrite paths to be extended, i.e., p is not empty, a tuple (ts, c) is popped from p. REST puts t, i.e., the last term of the path, into the set of output terms o and considers all terms t' that are: (a) not *already* in the path and (b) reachable by a single rewrite step of R (or returned by the function $\mathcal{E}$ explained later). The crucial decision of whether or not to extend a rewrite path with the additional step $t \to t'$ is handled in the **if** check of REST. This check is to guarantee termination: the *sat* check enforces that we only add rewrite steps which would leave the extended path still justifiable by *some* term ordering.

Figure 4 visualizes the rewrite paths explored by our algorithm for a run corresponding to the problem from Example 1, using the OCA for the recursive path quasi-ordering (Sec. 4.2)[3]. The manual proof in Example 1 corresponds to the right-most path in this tree; the other paths apply the same reasoning steps in different orders. In our implementation, we optimize the algorithm to avoid re-exploring the same term multiple times unless this could lead to further rewrites being discovered (cf. Sec. 6).

The arrow from the root of the tree to its child corresponds to the first rewrite REST applies: $f((s_0 \cup s_1) \cap s_0) \to f((s_0 \cap s_0) \cup (s_1 \cap s_0))$. This rewrite step can only be oriented by RPQOs with precedence $\cap > \cup$; therefore applying this rewrite constrains the set of RPQOs that REST must consider in subsequent applications. For example, the rewrite to the left child of $f((s_0 \cap s_0) \cup (s_1 \cap s_0))$ can only be oriented by RPQOs with precedence $\cup > \cap$. Since no RPQO can have both $\cap > \cup$ and $\cup > \cap$, no RPQO can orient the entire path from the root; REST must therefore reject the rewrite. On the other hand, the rewrite to the right child can be oriented by any RPQO where $s_0 > \emptyset$, $s_1 > \emptyset$, or $\cap > \emptyset$. The path from the root can thus continue down the right-hand side, as there are RPQOs that satisfy both $\cap > \cup$ and the other conditions. The subsequent rewrites down the right-hand side do not impose any new constraints on the ordering: $f((s_0 \cap s_0) \cup \emptyset) >_{\mathcal{T}} f(s_0 \cap s_0) >_{\mathcal{T}} f(s_0)$ in all RPQOs.

[3] We omit the commutativity rules from this run, just to keep the diagram easy to visualize, but our implementation handles the example easily with or without them.

Similarly, REST will prove Example 2 but will reject Example 3 when the input OCA represents RPQO orderings. As shown in our benchmarks (Table 2 of Sec. 7), Example 3 is solved by REST with an OCA for the Knuth-Bendix term ordering family.

3.3 Integrating an External Oracle

Finally, to tackle Challenge 5, we turn to the (so far ignored) third parameter of the algorithm, the external oracle $\mathcal{E}$. In the example variant presented at the end of Sec. 2, such a function might supply the rewrite step $s_0 \cap s_1 \to \emptyset$ by analysis of the logical assumption $|s_0 \cap s_1| = 0$, which goes beyond term-rewriting. More generally, any external solver capable of producing rewrite steps (equal terms) can be connected to our algorithm via $\mathcal{E}$. In our implementation in Liquid Haskell, we use the pre-existing *Proof by Logical Evaluation (PLE)* technique [52], which complements rewriting with the expansion of program function definitions, under certain checks made via SMT solving. Our only requirements on the oracle $\mathcal{E}$ are that it is bounded (finitely-branching) and strongly normalizing (cf. Sec. 5).

Our algorithm therefore flexibly allows the interleaving of term rewriting steps and external oracle steps; we avoid the potential for this interaction to cause non-termination by conditioning any further rewriting steps on the fact that the entire path (including the steps inserted by the oracle) can be oriented by at least one candidate term ordering.

The combination of our interface for defining term orderings via ordering constraint algebras, a search algorithm that effectively explores all rewrites enabled by the orderings, and the flexible possibility of combination with external solvers via the oracle parameter makes REST very adaptable and powerful in practice.

4 Well-Quasi-Orderings and the Ordering Constraint Algebra

Term orderings are typically defined as *strict well-founded* orderings; this requirement ensures that rewriting will obtain a normal form. However, as mentioned in Challenge 3, the restriction to strict orderings limits what can be achieved with rewriting. In this section we describe the derivation of well-quasi-orderings from strict orderings (Sec. 4.1) and introduce Knuth-Bendix quasi-orderings (Sec. 4.1.1) and recursive path quasi-orderings (Sec. 4.1.2), two novel term ordering families respectively based on the classical recursive path and Knuth-Bendix orderings. In addition, we formally introduce ordering constraint algebras (Sec. 4.2) and use them to develop an efficient ordering constraint algebra for RPQOs.

4.1 Well-Quasi-Orderings

We define well-quasi-orderings in the standard way.

▶ **Definition 5** (Well-Quasi-Orderings). *A relation $\geqslant$ is a quasi-order if it is reflexive and transitive. Given elements t and u in S, we say $t \approx u$ if $t \geqslant u$ and $u \geqslant t$. A quasi-order $\geqslant$ is also characterized as:*

1. *WQO, when for all infinite chains $x_1, x_2, \dots$ there exists an $i, j, i < j$ such that $x_j \geqslant x_i$,*
2. *thin, when forall $t \in S$, the set $\{u \in S \mid t \approx u\}$ is finite, and*
3. *total, when for all $t, u \in S$ either $t \geqslant s$ or $s \geqslant t$.*

Well-quasi-orderings are not required to be antisymmetric, however the corresponding strict part of the ordering must be well-founded. Hence, a WQO derives a strict ordering over equivalence classes of terms; REST also requires that these equivalence classes are finite (i.e., the ordering is thin). With this requirement, REST guarantees termination by exploring only

duplicate-free paths. Many simplification orderings can be converted into more permissive WQOs. Intuitively, given an ordering $>_o$ its quasi-ordering derivation also accepts equal terms, so we denote it as $\geqslant_o$. We next present two such derivations.

4.1.1 Knuth-Bendix Quasi-Orderings (KBQO)

The Knuth-Bendix ordering [31] is a well-known simplification ordering used in the Knuth-Bendix completion procedure. Here, we present a simplified version of the ordering, used by REST that is using ordering to only compare ground terms.

▶ **Definition 6.** *A weight function w is a function $\mathcal{F} \to \mathbb{N}$, where $w(f) > 0$ for all nullary functions symbols, and $w(f) = 0$ for at most one unary function symbol. w is compatible with a quasi-ordering $\geqslant_\mathcal{F}$ on $\mathcal{F}$ if, for any unary function f such that $w(f) > 0$, we have $f >_\mathcal{F} g$ for all g. $w(t)$ denotes the weight of a term t, such that $w(f(t_1, \ldots, t_n)) = w(f) + \sum_{1 \leqslant i \leqslant n} w(t_i)$.*

▶ **Definition 7** (Knuth-Bendix ordering (KBO) on ground terms)**.** *The Knuth-Bendix Ordering $>_{kbo}$ for a given weight function w and compatible precedence order $\geqslant_\mathcal{F}$ is defined as $f(t_1, \ldots, t_m) = t >_{kbo} u = g(u_1, \ldots, u_n)$ iff $w(t) \geqslant w(u)$, and 1) $w(t) > w(u)$, or 2) $f >_\mathcal{F} g$, or 3) $f \geqslant_\mathcal{F} g$ and $(t_1, \ldots, t_m) >_{kbolex} (u_1, \ldots, u_n)$. Where $>_{kbolex}$ performs a lexicographic comparison using $>_{kbo}$ as the underlying ordering.*

Intuitively, KBO compares terms by their weights, using $\geqslant_\mathcal{F}$ and the lexicographic comparison as "tie-breakers" for cases when terms have equal weights. However, as $\geqslant$ is already a well-quasi-ordering on $\mathbb{N}$, we can derive a more general ordering by removing these tie-breakers and the need for a precedence ordering at all.

▶ **Definition 8** (Knuth-Bendix Quasi-ordering (KBQO))**.** *Given a weight function w, the Knuth-Bendix quasi-ordering $\geqslant_{kbo}$ is defined as $t \geqslant_{kbo} u$ iff $w(t) \geqslant w(u)$.*

The resulting quasi-ordering is simpler to implement and more permissive: $t >_{kbo} u$ implies $t \geqslant_{kbo} u$; and also enables arbitrary associativity and commutativity axioms as rewrite rules, since it only considers the weights of the function symbols and no structural components of the term. One caveat is that REST operates on well-quasi-ordering that are thin (Def. 5), so it can only consider KBQOs with $w(f) > 0$ for all unary function symbols f.

However, the fact that KBO and KBQO largely ignore the structure of the term in their comparison has a corresponding downside: it is not possible to orient distributivity axioms, or many other axioms that increase the number of symbols in a term. Therefore, we have found that a WQO derived from the recursive path ordering [15] to be more useful in practice.

4.1.2 Recursive Path Quasi-Orderings (RPQO)

In this section, we define a particular family of orderings designed to be typically useful for term-rewriting via REST. Our family of orderings is a novel extension of the classical notion of RPO, designed to also be more compatible with symmetrical rules such as commutativity and associativity (cf. Challenge 3, Sec. 2).

Like the classical RPO notions, our *recursive path quasi-ordering* (RPQO) is defined in three layers, derived from an underlying ordering on function symbols:

- The input ordering $\succcurlyeq_\mathcal{F}$ can be any quasi-ordering over $\mathcal{F}$.

- The corresponding *multiset quasi-ordering* $\succcurlyeq_{M(X)}$ lifts an ordering $\succcurlyeq_X$ over X to an ordering $\succcurlyeq_{M(X)}$ over multisets of X. Intuitively $T \succcurlyeq_{M(X)} U$ when U can be obtained from T by replacing zero or more elements in T with the same number of equal (with respect to $\succcurlyeq_X$) elements, and replacing zero or more elements in T with a finite number of smaller ones (Def. 9).
- Finally, the corresponding *recursive path quasi-ordering* $\succcurlyeq_{rpo}$ is an ordering over terms. Intuitively $f(ts) \succcurlyeq_{rpo} g(us)$ uses $\succcurlyeq_{\mathcal{F}}$ to compare the function symbols f and g and the corresponding $\succcurlyeq_{M(rpo)}$ to compare the argument sets ts and us (Def. 10).

Below we provide the formal definitions of the multiset quasi-ordering and recursive path quasi-ordering respectively generalized from the multiset ordering of [18] and the recursive path ordering [15] to operate on quasi-orderings. For all the three orderings, we write $x_l < x_r \doteq x_l \not\succcurlyeq x_r$ and $x_l > x_r \doteq x_l \succcurlyeq x_r \wedge x_r \not\succcurlyeq x_l$.

▶ **Definition 9** (Multiset Ordering). *Given a ordering $\succcurlyeq_X$ over a set X, the derived multiset ordering $\succcurlyeq_{M(X)}$ over finite multisets of X is defined as $T \succcurlyeq_{M(X)} U$ iff: 1) $U = \emptyset$, or 2) $t \in T \wedge u \in U \wedge t \approx u \wedge (T - t) \succcurlyeq_{M(X)} (U - u)$, or 3) $t \in T \wedge (T - t) \succcurlyeq_{M(X)} (U \setminus \{u \in U \mid u <_X t\})$.*

▶ **Definition 10** (Recursive Path Quasi-Ordering). *Given a basic ordering $\succcurlyeq_{\mathcal{F}}$, the recursive path quasi-ordering (RPQO) is the ordering $\succcurlyeq_{rpo}$ over $\mathcal{T}$ defined as follows: $f(t_1, \ldots, t_m) \succcurlyeq_{rpo} g(u_1, \ldots, u_n)$ iff: 1) $f >_{\mathcal{F}} g$ and $\{f(t_1, \ldots, t_m)\} >_{M(rpo)} \{u_1, \ldots, u_n\}$, or 2) $g >_{\mathcal{F}} f$ and $\{t_1, \ldots, t_m\} \succcurlyeq_{M(rpo)} \{g(u_1, \ldots, u_n)\}$, or 3) $f \approx g$ and $\{t_1, \ldots, t_m\} \succcurlyeq_{M(rpo)} \{u_1, \ldots, u_n\}$.*

▶ **Example 11.** As a first example, any RPQO $\succcurlyeq_{\mathcal{T}}$ used to restrict term rewriting will accept the rule $X + Y \to Y + X$, since $X + Y \succcurlyeq_{\mathcal{T}} Y + X$ always holds. Since the top level function symbol is the same $+ \approx +$, by Def. 10 (3) we need to show $\{X, Y\} \succcurlyeq_{M(rpo)} \{Y, X\}$. By Def. 9 (2) (choosing both t and u to be X), we can reduce this to $\{Y\} \succcurlyeq_{M(rpo)} \{Y\}$; the same step applied to y reduces this to showing $\emptyset \succcurlyeq_{M(rpo)} \emptyset$ which follows directly from Def. 9 (1).

From this example, we can see that both $X + Y \succcurlyeq_{rpo} Y + X$ and $Y + X \succcurlyeq_{rpo} X + Y$ hold, in this case independently of the choice of input ordering $\succcurlyeq_{\mathcal{F}}$ on function symbols. In our next example, the choice of input ordering makes a difference.

▶ **Example 12.** As a next example, we compare the terms $s(X) + Y$ and $s(X + Y)$. Now that the outer function symbols are *not* equal, the order relies on the ordering between $+$ and s. Let's assume that $+ >_{\mathcal{F}} s$. Now to get $s(x) + y \succcurlyeq_{rpo} s(X + Y)$, the first case of Definition 10 further requires $\{s(X){+}Y\} >_{M(rpo)} \{X{+}Y\}$, which holds if $s(X){+}y >_{rpo} X{+}Y$. The outermost symbol for both expressions is $+$, so we must check the multiset ordering: $\{s(X), Y\} >_{M(rpo)} \{X, Y\}$, which holds because by case splitting on the relation between s and X, we can show that $s(X)$ is always greater than X. In short, if $+ >_{\mathcal{F}} s$, then $s(X) + Y \succcurlyeq_{rpo} s(X + Y)$.

Developing on our *RPQO* notion (Def. 10), we consider the set of *all* such orderings that are generated by any total, well-quasi-ordering over the operators. We prove that such term orderings satisfy the termination requirements of Theorem 22. Concretely:

▶ **Theorem 13.** *If $\succcurlyeq_{\mathcal{F}}$ is a total, well-quasi-ordering, then 1) $\succcurlyeq_{rpo}$ is a well-quasi-ordering, 2) $\succcurlyeq_{rpo}$ is thin, and 3) $\succcurlyeq_{rpo}$ is thin well-founded.*

4.2 Ordering Constraint Algebras

Ordering constraint algebras play a crucial role in the `REST` algorithm (Sec. 3.2), by enabling the algorithm to simultaneously consider an entire family of term orderings during the exploration of rewrite paths. In this section, we provide a formal definition for ordering constraint algebras and describe the construction of an algebra for the RPQO.

▶ **Definition 14** (Ordering Constraint Algebra). *An Ordering Constraint Algebra (OCA) over the set of terms T and term ordering family Γ, is a tuple $\mathcal{A}_{(T,\Gamma)} \doteq \langle C, \gamma, \top, \mathit{refine}, \mathit{sat}\rangle$, where:*

1. *C, the constraint language, can be any non-empty set. Elements of C are called constraints, and are ranged over by c.*
2. *γ, the concretization function of $\mathcal{A}_{(T,\Gamma)}$, is a function from elements of C to subsets of Γ.*
3. *$\top$, the top constraint, is a distinguished constant from C, satisfying $\gamma(\top) = \Gamma$.*
4. *refine, the refinement function, is a function $C \to T \to T \to C$, satisfying (for all c, t_l, t_r) $\gamma(\mathit{refine}(c, t_l, t_r)) = \{\succcurlyeq \mid \succcurlyeq \in \gamma(c) \wedge t_l \succcurlyeq t_r\}$.*
5. *sat, the satisfiability function, is a function $C \to \mathit{Bool}$, satisfying (for all c) $\mathit{sat}(c) = \mathit{true} \Leftrightarrow \gamma(c) \neq \emptyset$.*

The functions $\top$, *refine*, and *sat* are all called from our `REST` algorithm (Figure 3) and must be implemented as (terminating) functions to implement `REST`. Specifically, `REST` instantiates the initial path with constraints $c = \top$. When a path can be extended via a rewrite application $t_l \to_R t_r$, `REST` refines the prior path constraints c to $c' \doteq \mathit{refine}(c, t_l, t_r)$. Then, the new term is added to the path only if the new constraints are satisfiable ($\mathit{sat}(c')$ holds); that is, if c' admits an ordering that orients the generated path. The function γ need *not* be implemented in practice; it is purely a mathematical concept that gives semantics to the algebra.

Given terms T and a finite term ordering family Γ, a trivial OCA is obtained by letting $C = \mathcal{P}(\Gamma)$, and making γ the identity function; straightforward corresponding elements $\top$, *refine*, and *sat* can be directly read off from the constraints in the definition above.

However, for efficiency reasons (or in order to support potentially infinite sets of orderings, which our theory allows), tracking these sets symbolically via some suitably chosen constraint language can be preferable. For example, consider lexicographic orderings on pairs of constants, represented by a set T of terms of the form $p(q_1, q_2)$ for a fixed function symbol p and q_1, q_2 chosen from some finite set of constant symbols Q. We choose the term ordering family $\Gamma = \{\succcurlyeq_{lex(\succcurlyeq)} \mid \succcurlyeq \textit{ is a total order on } Q\}$ writing $\succcurlyeq_{lex(\succcurlyeq)}$ to mean the corresponding lexicographic ordering on $p(q_1, q_2)$ terms generated from an ordering $\succcurlyeq$ on Q.

A possible OCA over these T and Γ can be defined by choosing the constraint language C to be *formulas*: conjunctions and disjunctions of atomic constraints of the forms $q_1 > q_2$ and $q_1 = q_2$ prescribing conditions on the underlying orderings on Q. The concretization γ is given by $\gamma(c) = \{\succcurlyeq_{lex(\succcurlyeq)} \mid \succcurlyeq \textit{ satisfies } c\}$, i.e., a constraint maps to all lexicographic orders generated from orderings of Q that satisfy the constraints described by c, defined in the natural way. We define $\top$ to be e.g., $q = q$ for some $q \in Q$. A satisfiability function *sat* can be implemented by checking the satisfiability of c as a formula. Finally, by inverting the standard definition of lexicographic ordering, we define:

$$\mathit{refine}(c, p(q_1, q_2), p(r_1, r_2)) = c \wedge (q_1 > r_1 \vee (q_1 = r_1 \wedge q_2 > r_2))$$

Using this example algebra, suppose that `REST` explores two potential rewrite steps $p(a_1, a_2) \to p(b_1, a_2) \to p(a_1, a_1)$. Starting from the initial constraint $c_0 = \top$, the constraint for the first step $c_1 \doteq \mathit{refine}(c_0, p(a_1, a_2), p(b_1, a_2)) = a_1 > b_1 \vee (a_1 = b_1 \wedge a_2 > a_2)$ is satisfiable, e.g., for any total order for which $a_1 > b_1$. However, considering the subsequent step, the refined constraint $c_2 \doteq \mathit{refine}(c_1, p(b_1, a_2), p(a_1, a_1))$, computed as $c_2 = c_1 \wedge (a_2 >$

$a_2 \vee (a_2 = a_2 \wedge b_1 > a_1))$ is no longer satisfiable. Note that this allows us to conclude that there is no lexicographic ordering allowing this sequence of two steps, even without explicitly constructing any orderings.

We now describe an OCA for RPQOs (Sec. 4.1.2), based on a compact representation of sets of these orderings.

An Ordering Constraint Algebra for $\succcurlyeq_{rpo}$

The OCA for RPQOs enables their usage in REST's proof search. One simple but computationally intractable approach would be to enumerate the entire set of RPQOs that orient a path; continuing the path so long as the set is not empty. This has two drawbacks. First, the number of RPQOs grows at an extremely fast rate with respect to the number of function symbols; for example there are $6,942$ RPQOs describing five function symbols, and $209,527$ over six [29]. Second, most of these orderings differ in ways that are not relevant to the comparisons made by REST.

Instead, we define a language to succinctly describe the set of candidate RPQOs, by calculating the minimal constraints that would ensure orientation of the path of terms; REST continues so long as there is some RPQO that satisfies the constraints. Crucially the satisfiability check can be performed effectively using an SMT solver without actually instantiating any orderings.

Before formally describing the language, we begin with some examples, showing how the ordering constraints could be constructed to guide the termination check of REST.

▶ **Example 15** (Satisfiability of Ordering Constraints)**.** Consider the following rewrite path given by the rules $r_1 \doteq f(g(X), Y) \to g(f(X, X))$ and $r_2 \doteq f(X, X) \to f(k, X)$:

$$f(g(h), k) \to_{r_1} g(f(h, h)) \to_{r_2} g(f(k, h))$$

To perform the first rewrite REST has to ensure that there exists an RPQO $\succcurlyeq_{rpo}$ such that $f(g(h), k) \succcurlyeq_{rpo} g(f(h, h))$. Following from Definition 10, we obtain three possibilities:

1. $f >_{\mathcal{F}} g$ and $\{f(g(h), k)\} >_{M(rpo)} \{f(h, h)\}$, or
2. $g >_{\mathcal{F}} f$ and $\{g(h), k\} \succcurlyeq_{M(rpo)} \{g(f(h, h))\}$, or
3. $f \approx g$ and $\{g(h), k\} \succcurlyeq_{M(rpo)} \{f(h, h)\}$.

We can further simplify these using the definition of the multiset quasi-ordering (Def. 9). Concretely, the multiset comparison of (1) always holds, while the multiset comparisons of (2) and (3) reduce to $k >_{\mathcal{F}} f \wedge k >_{\mathcal{F}} g \wedge k >_{\mathcal{F}} h$. Thus, we can define the exact constraints c_0 on $\succcurlyeq_{rpo}$ to satisfy $f(g(h), k) \succcurlyeq_{rpo} g(f(h, h))$ as

$$c_0 \doteq f >_{\mathcal{F}} g \vee (k >_{\mathcal{F}} f \wedge k >_{\mathcal{F}} g \wedge k >_{\mathcal{F}} h)$$

Since there exist many quasi-orderings satisfying this formula (trivially, the one containing the single relation $f >_{\mathcal{F}} g$), the first rewrite is satisfiable.

Similarly, for the second rewrite, the comparison $g(f(z, z)) \succcurlyeq_{rpo} g(f(k, z))$ entails the constraints $c_1 \doteq z \succcurlyeq_{\mathcal{F}} k$. To perform this second rewrite the conjunction of c_0 and c_1 must be satisfiable. Since the second disjunct of c_0 contradicts c_1, the resulting constraints $f >_{\mathcal{F}} g \wedge z \succcurlyeq_{\mathcal{F}} k$ is satisfiable by an RPQO, thus the path is satisfiable.

▶ **Example 16** (Unsatisfiable Ordering Constraint)**.** As a second example, consider the rewrite rules $r_1 \doteq f(x) \to g(s(x))$ and $r_2 \doteq g(s(x)) \to f(h(x))$. These rewrite rules can clearly cause divergence, as applying rule r_1 followed by r_2 will enable a subsequent application of r_1 to a larger term. Now let's examine how our ordering constraint algebra can show the unsatisfiability of the diverging path:

$$f(z) \to_{r_1} g(s(z)) \not\to_{r_2} f(h(z))$$

$f(z) \succcurlyeq_{rpo} g(s(z))$ requires $c_0 \doteq f > g \wedge f > s$ which is satisfiable, but $g(s(z)) \succcurlyeq_{rpo} f(h(z))$ requires $c_1 \doteq (g \geqslant f \wedge g \geqslant h) \vee (g \geqslant f \wedge s \geqslant h) \vee (s > f \wedge s > h)$, which, although satisfiable, conflicts with c_0. Since no *RPQO* can satisfy both c_0 and c_1, the rewrite path is unsatisfiable.

Having primed intuition through the examples, we now present a way to compute such constraints. First, it is clear that we can define an RPQO based on the precedence over symbols $\mathcal{F}$. Therefore, we define our language of constraints to include the standard logical operators as well as atoms representing the relations between elements of $\mathcal{F}$, as:

$$C_{\mathcal{F}} \doteq f >_{\mathcal{F}} g \mid f \approx g \mid C_{\mathcal{F}} \wedge C_{\mathcal{F}} \mid C_{\mathcal{F}} \vee C_{\mathcal{F}} \mid \top \mid \bot$$

Next, we lift our definition of *RPQO* and the multiset quasi-ordering to derive functions: $rpo : \mathcal{T} \to \mathcal{T} \to C_{\mathcal{F}}$, and $mul : (\mathcal{T} \to \mathcal{T} \to C_{\mathcal{F}}) \to M(\mathcal{T}) \to M(\mathcal{T}) \to C_{\mathcal{F}}$. *rpo* is derived by a straightforward translation of Def. 10:

$$\begin{aligned} rpo(f(t_1,\ldots,t_m), g(u_1,\ldots,u_n)) = \quad & f >_{\mathcal{F}} g \quad \wedge \quad mul'(rpo, \{f(t_1,\ldots,t_m)\}, \{u_1,\ldots,u_n\}) \vee \\ & g >_{\mathcal{F}} f \quad \wedge \quad mul(rpo, \{t_1,\ldots,t_m\}, \{g(u_1,\ldots,u_n)\}) \vee \\ & f \approx g \quad \wedge \quad mul(rpo, \{t_1,\ldots,t_m\}, \{u_1,\ldots,u_n\}) \end{aligned}$$

where mul' is the strict multiset comparison: $mul'(f, T, U) = mul(f, T, U) \ \wedge \ \neg mul(f, U, T)$. $\neg : C_{\mathcal{F}} \to C_{\mathcal{F}}$ inverts the constraints, with $\neg(f >_{\mathcal{F}} g) = f \approx g \vee g >_{\mathcal{F}} f$ and $\neg(f \approx g) = f >_{\mathcal{F}} g \vee g >_{\mathcal{F}} f$; the other cases are defined in the typical way.

The definition for *mul* is more complex. Recall that $T \succcurlyeq_{M(X)} U$ when U can be obtained from T by replacing zero or more elements in T with the same number of equal (with respect to $\succcurlyeq_X$) elements, and by replacing zero or more elements in T with a finite number of smaller ones. Therefore each justification for $\{t_1,\ldots,t_m\} \succcurlyeq_{M(X)} \{u_1,\ldots,u_n\}$ can be represented by a bipartite graph with nodes labeled $t_1,\ldots,t_m$ and $u_1,\ldots,u_n$, such that:

1. Each node u_i has exactly one incoming edge from some node t_j.
2. If a node t_i has exactly one outgoing edge, it is labeled either `GT` or `EQ`.
3. If a node t_i has more than one outgoing edge, it is labeled `GT`.

$mul(f, \{t_1,\ldots,t_m\}, \{u_1,\ldots,u_n\})$ generates all such graphs: for each graph converts each labeled edge $(t, u, \texttt{EQ})$ to the formula $f(t, u) \wedge f(u, t)$, each edge $(t, u, \texttt{GT})$ to the formula $f(t, u) \wedge \neg f(u, t)$, and finally joins the formulas for the graph via a conjunction. The resulting constraint is defined to be the disjunction of the formulas generated from all such graphs.

Having defined the lifting of recursive path quasi-orderings to the language of constraints, we define our ordering constraint algebra $\mathcal{A}_{(\mathcal{T},\Gamma)}$ as the tuple $\langle C_{\mathcal{F}}, \top, refine, \gamma, sat\rangle$ where:

- $refine(c, t, u) = c \wedge rpo(t, u)$,
- Γ is the set of all RPQOs,
- $\gamma(c)$ is the set of RPQOs derived from the underlying quasi-orders $\succcurlyeq_{\mathcal{F}}$ that satisfy c, and
- $sat(c) = true$ if and only if there exists a quasi-order $\succcurlyeq_{\mathcal{F}}$ satisfying c.

That $\mathcal{A}_{(\mathcal{T},\Gamma)}$ is an OCA, i.e., satisfies the requirements of Def. 14, follows by construction. Namely, the function $rpo(t, u)$ produces constraints c such that, for any RPQO $\succcurlyeq_{rpo}$, $t \succcurlyeq_{rpo} u$ if and only if its underlying ordering $\succcurlyeq_{\mathcal{F}}$ satisfies c.

Having shown that using RPQOs as a term ordering is useful for theorem proving, satisfies the necessary properties for `REST`, and admits an efficient ordering constraint algebra, we continue our formal work by stating and proving the metaproperties of `REST`.

5 REST Metaproperties: Soundness, Completeness, and Termination

We now present the correctness, completeness, and termination of the REST algorithm defined in Figure 3. Here we only state the formal results; the detailed proofs can be found in [26]. Our formalism of rewriting is standard; based on that of Klop [30] (details in our extended version [26]). For a set of rewrite rules R, we $v \rightarrow_R w$ iff $v \rightarrow_r w$ for some $r \in R$. For oracle functions (from terms to sets of terms) $\mathcal{E}$, we write $t \rightarrow_{\mathcal{E}} t'$ *iff* $t' \in \mathcal{E}(t)$. We write $t \rightarrow_{R+\mathcal{E}} t'$ if $t \rightarrow_R t'$ or $t \rightarrow_{\mathcal{E}} t'$. For a relation $\rightarrow$ we write $\rightarrow^*$ for its reflexive, transitive closure. A *path* $t_1, \ldots, t_n$ is an indexed list of terms. A binary relation $\succcurlyeq$ *orients* a path $t_1, \ldots, t_n$ if $\forall i, 1 \leq i < n, t_i \succcurlyeq t_{i+1}$.

Soundness of REST means that any term of the output ($u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$) can be derived from the original input term by some combination of term rewriting steps from R and steps via the oracle function $\mathcal{E}$ (in other words, $t_0 \rightarrow^*_{R+\mathcal{E}} u$).

▶ **Theorem 17** (Soundness). *For all R, u, and t_0, if $u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$, then $t_0 \rightarrow^*_{R+\mathcal{E}} u$.*

Completeness of REST would naïvely be that, for any terms t_0 and u, if $t_0 \rightarrow^*_{R+\mathcal{E}} u$ then u is in our output ($u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$). This result doesn't hold by design, since REST explores only paths permitted by at least one instance of its input term ordering family. We prove this *relative* completeness result in two stages. First (Theorem 18), we show that completeness always holds if all steps only involve the external oracle. Second (Theorem 19), we prove relative completeness of REST with respect to the provided term ordering family.

▶ **Theorem 18** (Completeness w.r.t. $\mathcal{E}$). *For all R, u, t_0, if $t_0 \rightarrow^*_{\mathcal{E}} u$, then $u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$.*

▶ **Theorem 19** (Relative Completeness). *For all R, u, and t_0, if $t_0 \rightarrow^*_{R+\mathcal{E}} u$ and there exists an ordering $\succcurlyeq \in \gamma(\top)$ that orients the path justifying $t_0 \rightarrow^*_{R+\mathcal{E}} u$, then $u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$.*

Termination of REST requires conditions on the external oracle $\mathcal{E}$ and the ordering constraint algebra $\mathcal{A}$. Next, we formally define these requirements and state termination of REST.

▶ **Definition 20** (Well-Founded $\mathcal{A}$). *For ordering constraint algebras $\mathcal{A} = \langle C, \top, \mathit{refine}, \mathit{sat}, \gamma \rangle$, for $c, c' \in C$, we say c' strictly refines c (denoted $c' \sqsubset_{\mathcal{A}} c$) if $c' = \mathit{refine}(c, t, u)$ for some terms t and u, and $\gamma(c') \subset \gamma(c)$. Then, we say $\mathcal{A}$ is well-founded if $\sqsubset_{\mathcal{A}}$ is.*

Down every path explored by REST, the tracked constraint is only ever refined; well-foundedness of $\mathcal{A}$ guarantees that finitely many such refinements can be strict.

We note that if the OCA describes a finite set of orderings, then it is trivially well-founded: $\subset$ is well-founded on finite sets. For example, the ordering constraint algebra for RPQOs (Sec. 4.2) is well-founded when the set of functions symbols $\mathcal{F}$ is finite, as there are a only a finite number of possible RPQOs over a finite set of function symbols.

▶ **Definition 21** (Normalizing & Bounded $\mathcal{E}$). *A relation $t_l \rightarrow t_r$ is normalizing if it does not admit an infinite path and bounded if for each t_l it only admits finite t_r.*

Note that any deterministic, terminating external oracle is both normalizing and bounded.

▶ **Theorem 22** (Termination). *For any finite set of rewriting rules R, if: 1) $\rightarrow_{\mathcal{E}}$ is normalizing and bounded, 2) $\mathcal{A}$ is well-founded, and 3) the refine and sat functions of $\mathcal{A}$ are decidable (implemented to always-terminate), then, for all terms t_0, $\mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$ terminates.*

```
{-@ example1 :: s0 : Set → { s1 : Set | IsDisjoint s0 s1 } → f : (Set → a) → { f ((s0
      \/ s1) /\ s0)  = f s0 } @-}
example1 :: Set → Set → (Set → a) → Unit
example1 s0 s1 f =
       f ((s0 \/ s1) /\ s0)          ? distribUnion s0 s1 s0
   === f ((s0 /\ s0) \/ (s1 /\ s0)) ? idemInter s0
   === f (s0 \/ (s1 /\ s0))         ? symmInter s1 s0
   === f (s0 \/ (s0 /\ s1))         -- Disjoint
   === f (s0 \/ emptySet)           ? emptyUnion s0
   === f s0
   *** QED
```

Figure 5 Liquid Haskell version of the proof from Example 1.

6 Implementation of REST

We implemented REST as a standalone library, of 2337 lines of Haskell code (Sec. 6.1) and we integrated this library into Liquid Haskell [51] (Sec. 6.2) to automate *lemma* applications.

6.1 The REST Library

Our REST library is available on Hackage [25] and can be used directly by other Haskell projects. The library is designed modularly; for example, a client of the library can decide to use REST only for comparing terms via an OCA, without also using the proof search algorithm of Sec. 3.2. In addition, our library has a small code footprint and can be used with or without external solvers, making it ideal for integration into existing program analysis tools.

Furthermore, we include in the library built-in helper utilities for encoding and solving constraints on term orderings. Although the library enables integration of arbitrary solvers; it provides several built-in solvers for constraints on finite WQOs and also provides an interface for solving constraints with external SMT solvers. These utilities comprise the majority of the code in the REST library (1369 out of the 2337 lines).

Our implementation defines the OCA interface of Sec. 4.2 and provides three built-in instances for RPQOs, LPQOs (derived from the Lexicographic path ordering), and KBQOs (Sec. 4.1.1). The helper utilities included in the library enable a concise implementation of these OCAs: the three OCA implementations consist of 200 lines of code in total.

To facilitate debugging and evaluation of OCAs, the library also provides an executable that visualizes the rewrite paths that REST explores when using the OCA to compute the rewrite paths from a given term. For example, Figure 4 was produced using this functionality.

6.2 Integration of REST in Liquid Haskell

We used REST to automate lemma application in Liquid Haskell. Here we start with a brief overview of Liquid Haskell (Sec. 6.2.1), then present how REST is used to automate lemma instantiations (Sec. 6.2.2) and how it interacts with Liquid Haskell's automation (Sec. 6.2.3).

6.2.1 Liquid Haskell and Program Lemmas

Liquid Haskell performs program verification via *refinement types* for Haskell; function types can be annotated with refinements that capture logical/value constraints about the function's parameters, return value and their relation. For example, the function `example1` in Figure 5 ports the set example of Example 1 to Liquid Haskell, without any use of REST. User-defined lemmas amount to nothing more than additional program functions, whose refinement types

express the logical requirements of the lemma. The first line of the figure is special comment syntax used in Liquid Haskell to introduce refinement types; it expresses that the first parameter `s0` is unconstrained, while the second `s1` is refined in terms of `s0`: it must be some value such that `IsDisjoint s0 s1` holds. The refinement type on the (unit) return value expresses the proof goal; the body of the function provides the proof of this lemma. The proof is written in equational style; the `?` annotations specify lemmas used to justify proof steps [50]. The penultimate step requires no lemma; the verifier can discharge it based on the refinement on the `s1` parameter.

6.2.2 REST for Automatic Lemma Application in Liquid Haskell

We apply REST to automate the application of equality lemmas in the context of Liquid Haskell. The basic idea is to extract a set of rewrite rules from a set of refinement-typed functions, each of which must have a refinement type signature of the following shape:

```
{-@ rrule :: x₁:t₁ → ... → xₙ:tₙ → {v:() | e_l = e_r } @-}
```

In particular, the equality $e_l = e_r$ refinement of the (unit) return value generates potential rewrite rules to feed to REST, in both directions. Let $FV(e)$ be the free variables of e, if $FV(e_r) \subseteq FV(e_l)$ and $e_l \notin \{x_1, \ldots, x_n\}$ then $e_l \to e_r$ is generated as a rewrite rule. Symmetrically, if $FV(e_l) \subseteq FV(e_r)$ and $e_r \notin \{x_1, \ldots, x_n\}$ then $e_r \to e_l$ is generated as a rewrite rule. These rewrite rules are fed to REST along with the current terms we are trying to equate in the proof goal; any rewrites performed by REST are fed back to the context of the verifier as assumed equalities.

REST is using Liquid Haskell to ensure that the rewrite rules are correct. The body of `rrule` provides an proof (machine-checked by Liquid Haskell) that the equality $e_l = e_r$ holds. Such proofs can themselves use REST's rewrites, but mutual dependencies are not permitted, e.g., if $\texttt{rrule}_1$ is proved using $\texttt{rrule}_2$, then $\texttt{rrule}_2$'s proof cannot use $\texttt{rrule}_1$.

Selective Activation of Lemmas: Local and Global Rewrite Rules. In our Liquid Haskell extension, the user can activate a rewrite rule globally or locally, using the `rewrite` and `rewriteWith` pragmas, *resp.*. For example, with the below annotations

```
  {-@ rewrite global @-}   {-@ rewriteWith theorem [local] @-}
```

the rule `global` will be active when verifying every function in the current Haskell module, while the rule `local` is used only when verifying `theorem`.

Lemma Automation. Using our implementation, the same Example 1 proven manually in Figure 5 can be alternatively proven (with all relevant rewrite rules in scope) as follows:

```
{-@ example1 :: s0 : Set → { s1 : Set | IsDisjoint s0 s1 } →  f : (Set → a)
             → { f ((s0 \/ s1) /\ s0) = f s0 } @-}
example1 s0 s1 _ = ()
```

The proof is fully automatic: all required lemmas are handled by REST. Integrating REST into Liquid Haskell required around 500 lines of code, mainly for surface syntax.

6.2.3 Mutual PLE and REST Interaction

Liquid Haskell includes the *Proof by Logical Evaluation* (PLE) [52] tactic that automatically expands terminating functions. PLE expands function calls into single cases of their (possibly conditional) bodies exactly when the SMT can prove that a unique case definitely applies. In our implementation, PLE plays the role of its external oracle (cf. Sec. 3). Since PLE is proven terminating [52], the termination of this collaboration is also guaranteed (cf. Sec. 5).

Table 1 Comparison of REST with existing theorem provers. LH+ is Liquid Haskell with rewriting. The potential outcomes are ✓, followed by the runtime, when the property is proved; **loop** when no answer is returned after 300 sec; and **fail** when the property cannot be proven. Isa+ is Isabelle/HOL with Sledgehammer.

Property	LH+	Coq	Agda	Lean	Isabelle	Zeno	Isa+
Diverge	✓0.62s	loop	loop	fail	loop	✓0.47s	✓7.58s
Plus AC	✓1.13s	loop	loop	fail	fail	✓0.54s	✓4.30s
Congruence	✓0.69s	✓0.22s	✓26.10s	✓0.36s	✓3.86s	fail	✓4.39s

PLE takes as input a set $\mathcal{F}$ of (provably) terminating, user-defined function definitions that it iteratively evaluates. Meanwhile, REST is provided with the rewrite rules extracted from in-scope lemmas in the program (cf. Sec. 6.2.2); these two techniques can then generate paths of equal terms including steps justified by each technique. For example, consider the following simple lemma `countPosExtra`, stating that the number of strictly positive values in `xs ++ [y]` is the number in `xs`, provided that `y <= 0`, and a lemma stating that *countPos* of two lists appended gives the same result if their orders are swapped.

```
{-@ lm :: xs : [Int] → ys : [Int] → { countPos (xs ++ ys) = countPos (ys ++ xs) } @-}

{-@ rewriteWith countPosExtra [lm] @-}
{-@ countPosExtra :: xs : [Int] → {y : Int | y <= 0 } →
                    { countPos (xs ++ [y]) = countPos xs } @-}
countPosExtra :: [Int] → Int → ()
countPosExtra _ _ = () -- proof is fully automatic!
```

The proof requires rewriting `countPos(xs ++ [y])` first via lemma `lm` (by REST), expanding the definition of `++` twice (via PLE) to give `countPos(y:xs)`, and finally one more PLE step evaluating `countPos`, using the logical fact that `y` is not positive. Note that the first step requires applying an external lemma (out of scope for PLE) and the last requires SMT reasoning not expressible by term rewriting. The two techniques together allow for a fully automatic proof.

7 Evaluation

Our evaluation seeks to answer three research questions:

§ 7.1: How does REST compare to existing rewriting tactics?

§ 7.2: How does REST compare to E-matching based axiomatization?

§ 7.3: Does REST simplify equational proofs?

We evaluate REST using the Liquid Haskell implementation described in Sec. 6. In Sec. 7.1, we compare our implementation's rewriting functionality with that of other theorem provers, with respect to the challenges mentioned in Sec. 2. In Sec. 7.2, we compare against Dafny [35] by porting Dafny's calculational proofs to Liquid Haskell. Finally, in Sec. 7.3, we port proofs from various sources into Liquid Haskell both with and without rewriting, and compare the performance and complexity of the resulting proofs.

7.1 Comparison with Other Theorem Provers

To compare REST with the rewriting functionality of other theorem provers, we developed three examples to test the five challenges described in Sec. 2 and compare our implementation to that of other solvers. We chose to evaluate against Agda [41], Coq [11], Lean [5], Isabelle/HOL [40],

and `Zeno` [46], as they are widely known theorem provers that either support a rewrite tactic, or use rewriting internally. `Agda`, `Lean`, and `Isabelle/HOL` allow user-defined rewrites. In `Lean` and `Isabelle/HOL`, the tactic for applying rewrite rules multiple times is called `simp`; for simplification. `Agda`, `Coq`, and `Isabelle/HOL`'s implementation of rewriting can diverge for nonterminating rewrite systems [11, 1, 40]. On the other hand, `Lean` enforces termination, at least to some degree, by ensuring that associative and commutative operators can only be applied according to a well-founded ordering [4]. `Zeno` [46] does not allow for user-defined rewrite rules, rather it generates rewrites internally based on user-provided axioms. `Sledgehammer` [38, 44, 43] is a powerful tactic supported by `Isabelle/HOL` that (on top of the built-in rewriting) dispatches proof obligations to various external provers and succeeds when any of the external provers succeed; this tactic operates under a built-in (customizable) timeout.

1. `Diverge` tests how the prover handles the challenges 1 and 5: restricting the rewrite system to ensure termination and integrating external oracle steps. This example encodes a single (terminating) rewrite rule $f(X) \to g(s(s(X)))$ and terminating, mutually recursive definitions for f and g. However, the combination of the rules and function expansions can cause divergence. The proof follows directly from the function definitions.
2. `Plus AC` tests the challenges 2 and 3 by encoding a task that requires a permissive term ordering. This example encodes `p`, `q`, and `r`, user-defined natural numbers, and requires that expressions such as `(p + q) + r` can be rewritten into different groupings such as `(r + q) + p`, via associativity and commutativity rules.
3. `Congruence` is an additional test to ensure that the implementation of the rewrite system is permissive enough to generate the expected result. This test evaluates a basic expected property, that the expressions $f(g(t))$ and $f(g'(t))$ can be proved equal if there exists a rewrite rule of the form $g(X) \to g'(X)$.

We present our results in Table 1. As expected, `Coq`, `Agda`, and `Isabelle/HOL` diverge on the first example, as they do not ensure termination of rewriting. `Lean` does not diverge, but it also fails to prove the theorem. Unsurprisingly, the commutativity axiom of `Plus AC` causes theorem provers that don't ensure termination of rewriting to loop. Although `Lean` ensures termination, it does not generate the necessary rewrite application in every case, because it orients associative-commutative rewriting applications according to a fixed order. With the exception of `Zeno`, all of the theorem provers tested were able to prove the necessary theorem for the final example. Our implementation succeeds on these three examples by implementing a permissive termination check based on non-strict orderings.

The only two tools that proved all three examples are our implementation and Isabelle's Sledgehammer. The latter combines many techniques which go beyond term rewriting. Nonetheless, our novel approach provides a clear and general formal basis for incorporation with a wide variety of verifiers and reasoning techniques (due to its generic definition and formal requirements) and comes with strong formal guarantees for such combinations. In particular, `REST` guarantees termination and relative completeness, which Sledgehammer (via its timeout mechanism) does not.

7.2 Comparison with E-matching

To evaluate `REST` against the E-matching based approach, we compared with Dafny [35], a state-of-the-art program verifier. Dafny supports equational reasoning via calculational proofs [36] and calculation with user-defined functions [2]. We ported the calculational proofs of [36] to Liquid Haskell, using rewriting to automatically instantiate the necessary axioms.

7.2.1 List Involution

Figure 6 shows an example taken directly from Dafny [36], proving that the reverse operation on lists is an involution, i.e., $\forall$xs.*reverse(reverse(xs)) = xs*. In this example, both Liquid Haskell and Dafny operate on inductively defined lists with user-defined functions $+\!\!+$ and `reverse`. The original Dafny proof goes through via the combination of a manual application of a lemma called `ReverseAppendDistrib` (stating that for all lists xs and ys, $reverse(xs +\!\!+ ys) = reverse(ys) +\!\!+ reverse(xs)$) and induction on the size of the list.

Using `REST`'s term rewriting, Liquid Haskell is able to simplify the proof, with PLE expanding the function definitions for `reverse` and `append`, and `REST` applying the necessary equality `reverse (reverse xs ++ [x]) = reverse [x] ++ reverse (reverse xs)`.

In Dafny, a similar simplification of the calculational proof is not possible; the proof fails if the manual equality steps are simply removed. We experimented further and found that the lemma `ReverseAppendDistrib` can be alternatively encoded as a user-defined axiom which, by itself, does not appear to cause trouble for E-matching, and with this change alone the proof succeeds without the need for this single lemma call. On the other hand, the equalities must still be mentioned for the calculational proof to succeed. Perhaps surprisingly, removing these intermediate equality steps caused Dafny to stall[4]; analysis with the Axiom Profiler [7] indicated the presence of a (rather complex) matching loop involving the axiom `ReverseAppendDistrib` in combination with axioms internally generated by the verifier itself. This illustrates that achieving further automation of such E-matching-based proofs is not straightforward, and can easily lead to performance difficulties due to matching loops which can be hard to predict and understand, even in this state-of-the-art verifier. By contrast, `REST` can automatically provide the necessary equality steps without risking divergence.

7.2.2 Set Properties

Figure 7 shows the Dafny and Liquid Haskell proofs for the implication $s_0 \cap s_1 = \emptyset \implies f((s_0 \cup s_1) \cap s_0) = f(s_0)$. Dafny uses a calculational proof to show the equality $(s_0 \cup s_1) \cap s_0 = s_0$, seemingly by applying distributivity. In fact, the distributivity aspect is not relevant to the proof; rather, the set equality in the proof syntax causes Dafny to instantiate the set extensionality axiom discharging the proof. It is for this reason that Dafny requires an extra proof step to prove $f((s_0 \cup s_1) \cap s_0) = f(s_0)$, as this term does not include an equality on sets, but rather on applications of f. Dafny's set axiomatization does not include the distributivity axiom, as such an axiom could easily lead to matching loops.

`REST`'s termination property allows arbitrary lemmas to be encoded as rewrite rules; in this case rewriting with the distributivity lemma can complete the proof.

In conclusion, we have shown that `REST`'s rewriting can be used as an alternative to E-matching based axiomatization. Furthermore, the termination guarantee of `REST` enables axioms that may give rise to matching loops to, instead, be encoded as rewrite rules.

7.3 Simplification of Equational Proofs

Finally, we evaluate how `REST` can simplify equational proofs. We chose to include the set example from [36] (described in Sec. 7.2.2), data structure proofs from [50], examples from the Liquid Haskell test suite, as well as our own case study. We developed each example in Liquid Haskell both with and without rewriting, and compared the timing and proof

[4] We include this version in the Appendix of our extended paper [26].

```
lemma LemmaReverseTwice(xs: List)
    ensures reverse(reverse(xs)) == xs;
{
  match xs {
    case Nil =>
    case Cons(x, xrest) =>
      calc {
        reverse(reverse(xs));
        reverse(append(reverse(xrest), Cons(x, Nil)));
        { ReverseAppendDistrib(reverse(xrest), Cons(x, Nil)); }
        append(reverse(Cons(x, Nil)), reverse(reverse(xrest)));
        { LemmaReverseTwice(xrest); }
        append(reverse(Cons(x, Nil)), xrest);
        append(Cons(x, Nil), xrest);
        xs;
      }
  }
}
```

(a) Calculation-style proof in Dafny, from [36].

```
{-@ involutionP :: xs:[a] → {reverse (reverse xs) == xs } @-}
{-@ rewriteWith involutionP [distributivityP] @-}
involutionP []     = ()
involutionP (x:xs) = involutionP xs
```

(b) An equivalent proof implemented in Liquid Haskell extended with `REST`.

Figure 6 List Involution proofs in Liquid Haskell and Dafny.

complexity. Each proof using rewriting was evaluated using each different ordering constraint algebras built-in to our Haskell `REST` library. The proofs in [50] were selected because they require induction, expansion of user-defined functions, and equational reasoning steps to prove properties about trees and lists. The examples from the Liquid Haskell test suite were taken to evaluate the rewriting across a range of representative proofs.

Our `DSL` case study evaluates the performance of our implementation using a larger set of rewrite rules, by verifying optimizations for a simple programming language, containing statements (i.e., print, sequence, branches, repeats and no-ops) and expressions (i.e., constants, variables, arithmetic and boolean expressions) using 23 rewrite rules. Our rewriting technique can prove the kind of equivalences used in techniques such as supercompilation [8, 54, 48], by encoding the basic equality axioms as rewrite rules and using them to prove more complicated theorems. A full list of the axioms and proved theorems are available in our extended version [26]. We note that we encoded arithmetic operations as uninterpreted SMT functions, so that the built-in arithmetic theory of the SMT does not aid proof automation.

We present our results in Table 2. By using rewriting, we were able to eliminate all but two of the non-inductive axiom instantiations, while maintaining a reasonable verification time. As expected, no ordering constraint algebra was able to complete all the proofs using rewriting; however, each proof could be verified with at least one of them.

The test cases `LH-FingerTree` and `LH-MapReduce` required manual axiom instantiations because the structure of the term did not match the rewrite rule for the axiom. `LH-MapReduce`, requires proving the identity `op (f (take n is)) (mapReduce n f op (drop n is)) = f is`. An inductive lemma application generates the background equality `mapReduce n f op (drop n is) = f (drop n is)`, and a rewrite matching the term `op (f (take n is)) (f (drop n is))` must be instantiated to complete the proof. However, since the background equality

```
lemma Proof<a>(s0: set<int>, s1: set<int>, f: set<int> → a)
   requires s0 * s1 == {}
   ensures f((s0 + s1) * s0) == f(s0) {
     calc {
       (s0 + s1) * s0; (s0 * s0) + (s1 * s0);
       s0;
     }
   }
```

(a) Proof in Dafny using built-in set axiomatization.

```
{-@ assume unionEmpty     :: ma : Set → {v : () | ma \/ emptySet = ma } @-}
{-@ assume intersectComm  :: ma : Set → mb : Set → {v : () | ma /\ mb = mb /\ ma } @-}
{-@ assume intersectSelf  :: s0 : Set → { s0 /\ s0 = s0 } @-}
{-@ assume unionIntersect :: s0 : Set → s1 : Set → s2 : Set
                          → { (s0 \/ s1) /\ s2 = (s0 /\ s2) \/ (s1 /\ s2) } @-}
{-@ rwDisjoint :: s0 : Set → {s1 : Set | IsDisjoint s0 s1} → { s0 /\ s1 = emptySet } @-}

{-@ example1 :: s0 : Set → { s1 : Set | IsDisjoint s0 s1 } →  f : (Set → a) →
                { f ((s0 \/ s1) /\ s0) = f s0 } @-}
example1 s0 s1 _ = ()
```

(b) An equivalent proof implemented in Liquid Haskell, with a user-defined axiomatization of sets.

Figure 7 Set Proofs in Liquid Haskell and Dafny.

is neither a rewrite rule nor an evaluation step, the necessary term `op (f (take n is)) (f (drop n is))` never appears. Therefore, it is necessary to manually instantiate the lemma. As future work, a limited form of E-matching [12] could address this issue in the general case.

In conclusion, we've shown that extending Liquid Haskell to use `REST` enables rewriting functionality not subsumed by existing theorem provers, that `REST` is effective for axiom instantiation, and that `REST` can simplify equational proofs.

8 Related Work

Theorem Provers & Rewriting. Term rewriting is an effective technique to automate theorem proving [27] supported by most standard theorem provers. § 7.1 compares, by examples, our technique with `Coq`, `Agda`, `Lean`, and `Isabelle/HOL`. In short, our approach is different because it uses user-specified rewrite rules to derive, in a terminating way, equalities that strengthen the SMT-decidable verification conditions required for program verification.

SMT Verification & Rewriting. Our rewrite rules could be encoded in SMT solvers as universally quantified equations and instantiated using *E-matching* [12], i.e., a common algorithm for quantifier instantiation. Without careful choice of user-specified *triggers*, E-matching can lead to hard-to-predict an unstable performance, including non-termination due to axioms generating new instantiations indefinitely in a *matching loop*. [34] refers to this unpredictable behavior of E-matching as the "the butterfly effect" and partially addresses it by detecting formulas that could give rise to simple matching loops. However, as we show in Sec. 7.2.1, guaranteeing termination in general remains subtle, fundamentally due to the fact that every equality generates a (potentially-infinite) equivalence class of terms available in the solver's search. Our approach circumvents unpredictability by using the terminating `REST` algorithm to instantiate the rewrite rules outside of the SMT solver.

Table 2 Results from simplification of proofs with rewriting. **Set-Dafny** is the set example from[36], **Set-Mono** describes a similar property. **List** and **Tree** are equational proofs from [50]. **DSL** is the program equivalence case study. The remaining proofs are from the Liquid Haskell test suite folder `tests/pos`, excluding those using only inductive or mutually inductive lemmas. **Orig.** is the number of non-inductive lemma applications in the original proof. **Cut** is the number of lemma applications that were removed by rewriting; where **Cut** is the same as **Orig.**, all non-inductive lemma applications have been removed. **Rules** is the number of axioms encoded as rewrite rules. **Time (Orig.)** is verification time in seconds for the original proof. **LPQO** and **KBQO** are OCAs derived from the Lexicographic Path Ordering and Knuth-Bendix ordering respectively, and **Fuel** is an OCA allowing up to 5 rewrite applications per proof goal.

Name	Orig.	Cut	Rules	Time				
				Orig.	RPQO	LPQO	KBQO	Fuel
Set-Dafny	4	4	5	1.11s	✓1.15s	✓1.19s	✗1.13s	✓1.22s
Set-Mono	7	7	4	1.16s	✗1.40s	✗1.41s	✓1.47s	✓1.60s
List	3	3	3	2.46s	✓3.17s	✗4.21s	✗2.24s	✓3.54s
Tree	3	3	3	1.61s	✓2.64s	✓3.40s	✓3.08s	✓3.12s
DSL	43	43	23	2.89s	✓5.46s	✗3.85s	✗4.19s	✓6.54s
LH-FingerTree	2	1	1	5.55s	✓5.60s	✓5.57s	✓5.64s	✓5.95s
LH-T1013	1	1	1	1.11s	✓1.06s	✓1.00s	✓1.02s	✓1.06s
LH-T1025	2	2	2	1.03s	✓1.05s	✓1.08s	✓1.07s	✓1.13s
LH-T1548	1	1	1	1.45s	✓1.33s	✓1.38s	✓1.32s	✓1.45s
LH-T1660	1	1	1	1.09s	✓1.12s	✓1.12s	✓1.12s	✓1.20s
LH-MapReduce	4	3	2	14.38s	✓29.50s	✓518.91s	✓28.49s	✗Timeout

Z3 [13] and `CVC4` [6] are state-of-the-art SMT solvers; both support theory-specific rewrite rules internally. Recent work [42] enables user-provided rewrite rules to be added to `CVC4`. However, using the SMT solver as a rewrite engine offers little control over rewrite rule instantiation, which is necessary for ensuring termination.

Rewriting in Haskell. Haskell itself has used various notions of rewriting for program verification. GHC supports the `RULES` pragma with which the user can specify unchecked, quantified expression equalities that are used at compile time for program optimization. [10] proposes Inspection Testing as a way to check such rewrite rules using runtime execution and metaprogramming, while [22] prove rewrite rules via metaprogramming and user-provided hints. In a work closely related to ours, Zeno [46] is using rewriting, induction, and further heuristics to provide lemma discovery and fully automatic proof generation of inductive properties. Unlike our approach, Zeno's syntax is restricted (e.g., it does not allow for existentials) and it does not allow for user-provided hints when automation fails. HALO [53] enables Haskell verification by converting Haskell into logic and using an SMT solver to verify user-defined formulas. However, this approach relies on SMT quantifiers to encode user functions, thus the solver can diverge and verification becomes unpredictable.

Termination of Rewriting and Runtime Termination Checking. Early work on proving termination of rewriting using simplification orderings is described in [15]. More recent work involves dependency pairs [3] and applying the size-change termination principle [33] in the context of rewriting [49]. Tools like AProVE [24] and NaTT [56] can statically prove the termination of rewriting. In contrast, `REST` is not focused on statically proving termination of rewriting; rather it uses a well-founded ordering to ensure termination at runtime. This

approach enables integration of arbitrary external oracles to produce rewrite applications, as a static analysis is not possible in principle. Furthermore, our approach enables nonterminating rewriting systems to be useful: REST will still apply certain rewrite rules to satisfy a proof obligation, even if the rewrite rules themselves cannot be statically shown to terminate.

We used a well-quasi-ordering [32] because it enables rewriting to terms that are not strictly decreasing in a simplification ordering. WQOs are commonly used in online termination checking [37], especially for program optimization techniques such as supercompilation [9].

Equality Saturation. In our implementation, REST passes equalities to the SMT environment, ultimately used for *equality saturation* via an E-graph data structure [20]. Equality saturation has also been used for supercompilation[48]. REST does not currently exploit equality saturation (unless indirectly via its oracle). However, as future work we might explore local usage of efficient E-graph implementations. (e.g., [55]) for caching the equivalence classes generated via rewrite applications.

Associative-Commutative Rewriting. Traditionally, enforcing a strict ordering on terms prevents the application of rewrite rules for associativity or commutativity (AC); this problem motivates REST's use of well-quasi orders. However, another solution is to omit the rules and instead perform the substitution step of rewriting modulo AC. Termination of the resulting system can be proved using an AC ordering [17]; the requirement is that the ordering respects AC: for all terms t' AC-equivalent to t and u' AC-equivalent to u, $t > u$ implies $t' > u'$.

REST's use of well-quasi-orderings enables AC axioms to be encoded as rewrite rules, guaranteeing completeness if the AC-equivalence class of a term is a subset of the equivalence class induced by the ordering. This is a significant practical benefit as it does not require REST to identify AC symbols and treat them differently for unification.

However, treating AC axioms as rewrite rules can lead to an explosion in the number of terms obtained via rewriting. As future work, it could be possible to extend REST to support AC rewriting and unification in order to reduce the number of explicitly instantiated terms.

9 Conclusion

We presented REST, a novel approach to rewriting that uses an online termination check that simultaneously considers entire families of term orderings via Ordering Constraint Algebras. We defined our algebra on well-quasi orderings that are more permissive than standard simplification orderings and demonstrated how to derive well-quasi orderings from well-known simplification orderings. We proved correctness, relative completeness, and (online) termination of REST and implemented it as a small Haskell library suitable for integration with existing verification tools. To evaluate REST we integrated our implementation with Liquid Haskell and showed that the resulting system compares well with existing rewriting techniques and can substantially simplify equational proofs.

References

1 Agda Developers. *The Agda Language Reference, version 2.6.1*, 2020. Available electronically at `https://agda.readthedocs.io/en/v2.6.1/language/index.html`.

2 Nada Amin, K Rustan M Leino, and Tiark Rompf. Computing with an smt solver. In *International Conference on Tests and Proofs*, pages 20–35. Springer, 2014.

3 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1):133–178, April 2000. doi:10.1016/S0304-3975(99)00207-8.

4 Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean, Release 3.20.0*, September 2020. p 73. URL: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

5 Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich. *The Lean Reference Manual, Release 3.3.0*, 2018. URL: https://leanprover.github.io/reference/lean_reference.pdf.

6 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah. URL: http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf.

7 N. Becker, P. Müller, and A. J. Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2019*, LNCS, pages 99–116. Springer-Verlag, 2019.

8 Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. *SIGPLAN Not.*, 45(11):135–146, September 2010. doi:10.1145/2088456.1863540.

9 Maximilian Bolingbroke, Simon Peyton Jones, and Dimitrios Vytiniotis. Termination combinators forever. In *Proceedings of the 4th ACM symposium on Haskell*, pages 23–34, 2011.

10 Joachim Breitner. A promise checked is a promise kept: inspection testing. In Nicolas Wu, editor, *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 14–25. ACM, 2018. doi:10.1145/3242744.3242748.

11 The Coq Development Team. *The Coq Reference Manual, version 8.11.2*, 2020. Available electronically at http://coq.inria.fr/refman.

12 Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

13 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

14 Nachum Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979. doi:10.1016/0020-0190(79)90071-1.

15 Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical computer science*, 17(3):279–301, 1982.

16 Nachum Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 3(1-2):69–115, 1987.

17 Nachum Dershowitz, Jieh Hsiang, N Alan Josephson, and David A Plaisted. Associative-commutative rewriting. In *IJCAI*, pages 940–944, 1983.

18 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 188–202, Berlin, Heidelberg, 1979. Springer. doi:10.1007/3-540-09510-1_15.

19 David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005. doi:10.1145/1066100.1066102.

20 David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

21 Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Adding decision procedures to smt solvers using axioms with triggers. *Journal of Automated Reasoning*, 56(4):387–457, 2016.

22 Andrew Farmer, Neil Sculthorpe, and Andy Gill. Reasoning with the hermit: Tool support for equational reasoning on ghc core programs. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 23–34, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2804302.2804303.

23 Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

24 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

25 Zachary Grannan. rest-rewrite: Rewriting library with online termination checking, 2022. URL: https://hackage.haskell.org/package/rest-rewrite.

26 Zachary Grannan, Niki Vazou, Eva Darulova, and Alexander J. Summers. Rest: Integrating term rewriting with program verification (extended version), 2022. arXiv:2202.05872.

27 Jieh Hsiang, Hélène Kirchner, Pierre Lescanne, and Michaël Rusinowitch. The term rewriting approach to automated theorem proving. *The Journal of Logic Programming*, 14(1):71–99, October 1992. doi:10.1016/0743-1066(92)90047-7.

28 Gerard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 30–45, USA, 1977. IEEE Computer Society. doi:10.1109/SFCS.1977.9.

29 OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2022. A000798: Number of different quasi-orders (or topologies, or transitive digraphs) with n labeled elements. URL: https://oeis.org/A000798.

30 J. W. Klop. *Term Rewriting Systems*, pages 1–116. Oxford University Press, Inc., USA, 1993.

31 Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

32 Joseph B Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory, Series A*, 13(3):297–305, November 1972. doi:10.1016/0097-3165(72)90063-5.

33 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 81–92, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.360210.

34 K. R. M. Leino and Clément Pit-Claudel. Trigger Selection Strategies to Stabilize Program Verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 361–381, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-41528-4_20.

35 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

36 K Rustan M Leino and Nadia Polikarpova. Verified calculations. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 170–190. Springer, 2013.

37 Michael Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566, pages 379–403. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. Series Title: Lecture Notes in Computer Science. doi:10.1007/3-540-36377-7_17.

38 Jia Meng and Lawrence C Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

39 P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, 2016.

40 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2020.

41 Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2008. Springer-Verlag.

42 Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. Syntax-guided rewrite rule enumeration for smt solvers. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 279–297, Cham, 2019. Springer International Publishing.

43 Lawrence C Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In *International Conference on Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.

44 Lawrence C Paulsson and Jasmin C Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *Proceedings of the 8th International Workshop on the Implementation of Logics (IWIL-2010), Yogyakarta, Indonesia. EPiC*, volume 2, 2012.

45 Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-c: a software analysis perspective. *Formal Aspects of Computing*, 27, October 2012. doi:10.1007/s00165-014-0326-7.

46 William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–421, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

47 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016. URL: https://www.fstar-lang.org/papers/mumon/.

48 Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480915.

49 René Thiemann and Jürgen Giesl. Size-Change Termination for Term Rewriting. In *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706, pages 264–278, March 2007. doi:10.1007/3-540-44881-0_19.

50 Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: equational reasoning in liquid Haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, pages 132–144, St. Louis, MO, USA, September 2018. Association for Computing Machinery. doi:10.1145/3242744.3242756.

51 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2628136.2628161.

52 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158141.

53 Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. Halo: Haskell to logic through denotational semantics. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 431–442, 2013.

54 Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990. `doi:10.1016/0304-3975(90)90147-A`.

55 Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

56 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Nagoya termination tool. In *Rewriting and Typed Lambda Calculi*, pages 466–475. Springer, 2014.

Static Analysis for AWS Best Practices in Python Code

Rajdeep Mukherjee
Amazon Web Services, San Jose, CA, USA

Omer Tripp
Amazon Web Services, San Jose, CA, USA

Ben Liblit
Amazon Web Services, Arlington, VA, USA

Michael Wilson
Amazon Web Services, Seattle, WA, USA

Abstract

Amazon Web Services (AWS) is a comprehensive and broadly adopted cloud provider. AWS SDKs provide access to AWS services through API endpoints. However, incorrect use of these APIs can lead to code defects, crashes, performance issues, and other problems. AWS best practices are a set of guidelines for correct and secure use of these APIs to access cloud services, allowing conformant clients to fully reap the benefits of cloud computing.

We present static analyses, developed in the context of a commercial service for detection of code defects and security vulnerabilities, to identify deviations from AWS best practices. We focus on applications that use the AWS SDK for Python, called *Boto3*. Precise static analysis of Python cloud applications requires robust type inference for inferring the types of cloud service clients. However, Boto3's "Pythonic" APIs pose unique challenges for type resolution, as does the interprocedural style in which service clients are used. We offer a layered approach that combines multiple type-resolution and tracking strategies in a staged manner: (i) general-purpose type inference augmented by type annotations, (ii) interprocedural dataflow analysis expressed in a domain-specific language, and (iii) name-based resolution as a low-confidence fallback. Across >3,000 popular Python GitHub repos that make use of the AWS SDK, our layered type inference system achieves 85% precision and 100% recall in inferring Boto3 clients in Python client code.

Additionally, we use real-world developer feedback to assess a representative sample of eight AWS best-practice rules. These rules detect a wide range of issues including pagination, polling, and batch operations. Developers have accepted more than 85% of the recommendations made by five out of eight Python rules, and almost 83% of all recommendations.

2012 ACM Subject Classification Theory of computation → Program analysis; Computer systems organization → Cloud computing

Keywords and phrases Python, Type inference, AWS, Cloud, Boto3, Best practices, Static analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.14

Related Version *Extended Version*: https://arxiv.org/abs/2205.04432

1 Introduction

Amazon Web Services (AWS) is a comprehensive and broadly adopted cloud provider. *AWS best practices* are a set of guidelines for correct, secure, and performant usage of AWS cloud SDKs. Python is used extensively to build applications on top of the AWS cloud, using the AWS SDK for Python, called *Boto3*. We report on our experience developing static analyses to enforce AWS best practices in Boto3-based Python applications. These rules are evaluated as part of a commercial cloud service, Amazon CodeGuru Reviewer (henceforth, *CodeGuru*) [9], that runs static analysis on customer code to detect security vulnerabilities, optimization opportunities, and other defects. Figure 1 shows the CodeGuru architecture.

© Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson;
licensed under Creative Commons License CC-BY 4.0

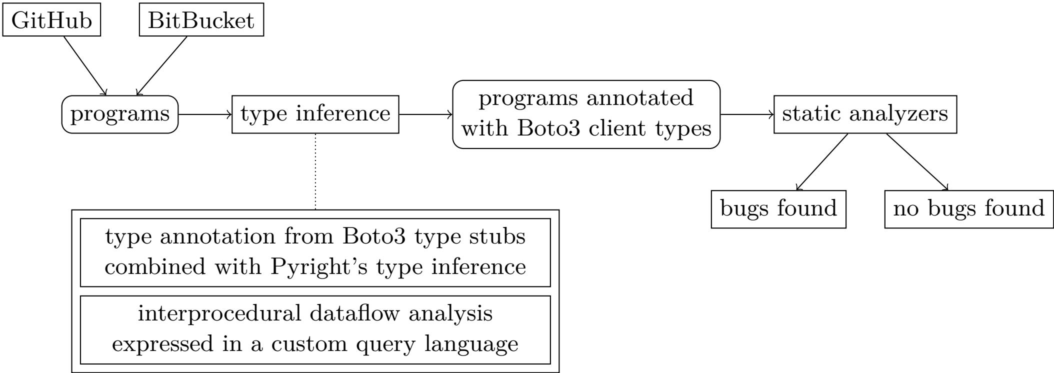


Figure 1 High-level overview of CodeGuru.

CodeGuru supports Java and Python, and integrates with different code hosting platforms including GitHub and BitBucket. CodeGuru supports three code scanning modes:

- Incremental: A code review is created automatically when a pull request is raised.
- Full: The entire codebase is analyzed.
- CI/CD: The entire codebase is analyzed as part of CI/CD workflows.

1.1 Importance of AWS Best Practices

Deviation from AWS best practices can lead to large-scale operational failures. Consequences include race conditions leading to service outage or auto-ticketing errors; authorization and authentication errors; broken throttling mechanisms that impose unexpected loads on services, thereby leading to high latency or timeouts; missing or incorrect error handling leading to billing errors; and many other severe problems. Risks are commonly discovered by manual inspection or testing. However, many such cases can be detected, and prevented, by applying static analysis to clients of the AWS SDK. The AWS best practices rules that we have developed alert developers to such defects during code review, before customer impact.

We provide lower-bound metrics to give an idea of CodeGuru's throughput. In an average week, CodeGuru analyzes $\gg 10,000$ pull requests (PRs) containing $\gg 1,000,000$ lines of code across $\gg 100,000$ files, and provides $\gg 1,000$ AWS best practices recommendations due to $\gg 100$ different static analysis detectors.

1.2 Scope

CodeGuru supports AWS best practices for both Java and Python. We focus on Python given its dynamic nature and lack of strict static typing. For precise enforcement of AWS best practices, it is essential to identify function calls into the AWS SDK, and which service in particular is used. Java reveals this information through static types, but in Python this information is not available by default: a challenging start for our analyses.

We describe several on-demand type resolution strategies and combinations thereof. We consider three core approaches: (1) Boto3 type stubs, in combination with general-purpose **type inference**, to resolve types when processing the Python AST; (2) on-demand interprocedural **dataflow tracking**, in both the forward and the backward directions, to check whether the receiver of a function call corresponds to a given AWS service; and (3) a lightweight over-approximation that simply checks whether the **called function's name** is compatible with a given AWS service's API. We present the approaches themselves and more

advanced algorithms that combine these approaches. We also provide technical details on the underlying infrastructure that enabled us to implement these approaches: CodeGuru's code representation and language for rule specification.

1.3 Main Contributions

This principal contributions of this paper are as follows. (1) We offer an on-demand type resolution strategy, which we demonstrate as effective in the case of Python clients of the AWS SDK. (2) In support of the above-mentioned strategy, we present the intermediate representation (IR) and query language used by the CodeGuru service. (3) We describe a representative sample of the AWS best practices rule suite running as part of the CodeGuru service. (4) We share our evaluation on 3,027 GitHub repositories, and real-world feedback we received from developers, to validate our approach.

1.4 Paper Structure

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3, we present background about the Boto3 SDK. Section 4 shows several examples that motivate the need for advanced type inference. Sections 5 and 6 lay down the technical infrastructure for our approach in describing, respectively, the code representation and query language that we use to express Python AWS best practices rules. Section 7 describes the different type inference capabilities we have developed, based both on Boto3 type stubs and data-flow tracking. In Section 8 we examine eight representative Python AWS best practices rules. Section 9 states our research hypotheses and reports on experiments to assess the efficacy of our type inference strategies. Section 10 concludes and outlines future research.

2 Related Work

Different approaches have been taken to infer Python type annotations, and formalize Python semantics more generally. We review approaches based on program analysis as well as machine learning, and compare these approaches with CodeGuru.

2.1 Classical Program Analysis

Widely used Python type checkers include mypy [25], Pyre [15], pytype [20], and Pyright [26]. These tools rely on manual type annotations provided by developers, augmented with varying forms of type inference. However, retrofitting type annotations onto large libraries or applications can be tedious and error-prone. Other prior work places more emphasis on static analysis [10, 16, 17, 22, 27, 32] or dynamic analysis [34] to reduce reliance on human-authored annotations. Our initial search for supporting infrastructure found that many published tools have failed to keep up with recent Python releases, or omit support for key Python features such as exceptions [16] or recursion [27]. We opted to use Pyright as our baseline, as Pyright is both actively maintained and has a rather advanced inference engine (see Section 7.1). In spite of these advantages, Pyright alone proved unsatisfactory for our cloud application domain. The details, as conveyed in Section 9, may serve to highlight challenges for other developers of general-purpose type inference engines.

When writing type annotations, Python developers often focus on function signatures: arguments and return values. Some research tools mirror this bias, such as TypeWriter [30]. Xu et al. [36] present a probabilistic type inference system, but the accuracy of probabilistically inferred types for Python variables is limited. Our work requires accurate types for variables, making these two approaches unsuitable.

Any attempt to statically analyze Python code must contend with the intricacies of the Python language. Notable efforts to formalize Python semantics include those by Smeding [33], Politz et al. [29] and Köhl [24]. Smeding's work predates Python type annotations, while neither Politz et al. nor Köhl mention them in any way. These omissions are not surprising, as type annotations have only limited effects on runtime behavior. Thus, these codifications of Python semantics offer little insight regarding the type-inference challenges addressed here. Our approach is neither sound nor complete (see Section 5.4), so a standard type-soundness theorem relating static types to runtime semantics does not apply.

In the specialized domain of machine learning, where Python is perhaps the most popular language, WALA Ariadne [13] analyzes Python specifically to infer the dimensions and types of tensors. Like Ariadne, our work is motivated by a specific application domain, and even a specific framework: Ariadne focuses on machine learning using TensorFlow [1]; CodeGuru focuses on cloud computing using Boto3. Ariadne's solution entails both a custom type system and an analysis to infer it. Our approach builds upon standard Python types and type annotations. While we crafted our analysis strategy to match idiomatic Boto3 use, these idioms are not exclusive to Boto3 client code. Therefore our layered approach may be more broadly applicable.

2.2 Machine Learning

PYInfer [12] uses deep learning to generate type annotations for Python. PYInfer fuses deep learning with static analysis such as PySonar2 to infer types for variables as well as function-level types in Python. All of these techniques either require labelled type annotations or employ a static analyzer to generate the initial annotations from Python repositories in order to train the deep neural network. However, type resolution for Boto3 service clients is non-trivial due to the reasons mentioned above.

JSNice [31], DeepTyper [23], and LambdaNet [35] use deep learning to generate type annotations for JavaScript and/or TypeScript. LambdaNet's authors note that TypeScript is an inviting target because "plenty of training data is available in terms of type-annotated programs." In principle, similar strategies may be applicable to Python. However, it is unclear whether the available corpus of type-annotated Python Boto3 client programs is large enough for effective training in practice.

3 Background on Boto3: the AWS SDK for Python

This section describes the AWS service clients in the AWS SDK for Python, also called "Boto3". [4]

3.1 Clients and Resources: Low- and High-Level APIs

Boto3 has two distinct levels of APIs:

Client (or "low-level") APIs provide one-to-one mappings to the underlying HTTP API operations.

Resource APIs hide explicit network calls but instead provide resource objects and collections to access attributes and perform actions. Resources represent an object-oriented interface to AWS. They provide a higher-level abstraction than the raw, low-level calls made by service clients.

A low-level service client can be created by passing the name of service as an argument to the boto3.client method. [7] For example, the Python statement, s3_client = boto3.client('s3'), creates a low-level client for the Amazon Simple Storage Service (S3). Conversely, a service resource can be created by passing the name of service as an argument to the SDK boto3.resource method. [8] For example, the Python statement, s3_client = boto3.resource('s3'), creates an Amazon S3 service resource. It is also possible to access the low-level client from an existing resource, as in:

```
s3_resource = boto3.resource('s3')
s3_client = s3_resource.meta.client
```

Alternatively, to use service resources, one can invoke the resource() method of a Session and pass in a service name. For example, one can create an Amazon S3 service resource using:

```
session = boto3.session.Session()
s3_resource = session.resource('s3')
```

Service clients give access to service operations by calling methods on a client. For example, suppose s3_client is an S3 client. Then one can create an S3 bucket, with the bucket name passed via an argument, using:

```
response = s3_client.create_bucket(Bucket=bucket_name)
```

3.2 Boto3 Type Stubs

Boto3-stubs provides full type annotations for Boto3. [14] In particular, Boto3-stubs provides annotations for a Client type, ServiceResource, and Resource type for each AWS service. It also provides annotations for a Waiter type, and a Paginator type for each service. With help from Boto3-stubs, several Python type-checking tools can discover types for multiple flavors of client construction calls such as boto3.client, boto3.session, session.client, and session.session.

3.3 API Specifications From Boto3

Some of the AWS best practice rules that are presented in this paper use an external configuration that provides a specification of some service-specific fragment of the complete Boto3 API. This specification includes an API name, type, the service name the API belongs to, and few other attributes that are relevant for the rule. We refer to these external configurations as *API specifications*. One such example is presented in Section 9. API specifications are automatically extracted from Boto3 API models. [6] These API models have specific traits, such as, *Pagination*, *Batch*, *Deprecated*, *Waiters*, or *mutual-exclusion*, which help determine the characteristics of the API. We extract relevant API traits from API models across Boto3 services to construct the complete API specification to enforce. These API specifications are then used by the best practice rules for analyzing client code.

4 Motivating Examples

This section presents an example that motivate the need for sophisticated type inference to recover the types of AWS service clients in real-world Python applications. The type annotations in Figure 2 are obtained from Pyright with Boto3 type stubs, which are on lines with the prefix “#→”.

```
import boto3

class Example(object):
  def get_sns_client():
    return boto3.resource("sns")

  def M1():
    sns_arn = os.environ['PUBLISH']
    client = get_sns_client()
    #→ client: SNSServiceResource
    M2(client, topic, subscription)
    return client.Topic(sns_arn)

  def M2(client, topic, subscription):
    topic = client.topic(topic)
    #→ (variable) client: Any
```

Figure 2 Example of a Python application code using Boto3.

▶ **Example 1.** Consider the Python code snippet in Figure 2. Here, the Boto3 client is returned by `get_sns_client()`. Its type is `SNSServiceResource`, marked in bold in method `M1`. This type correctly identifies `client` as a client for the Amazon Simple Notification Service (SNS). Figure 2 creates `client` using the `boto3.resource()` API which gives an object-oriented interface to SNS. [8]. The client flows into `M2` via a function parameter. `M2` uses `client` to make API call, `topic()`. Unfortunately, Pyright was unable to assign `client` a precise type, leaving it typed simply as the generic `Any` inside `M2`. Inference falls short here because Pyright cannot guarantee that `client` must *always* be an `SNSServiceResource` in *every* possible call to M2. This is safe but, for our purposes, unfortunate: an untyped `client` cascades into untyped `topic` and `subscription`, leaving us with nothing useful to analyze for any of the API calls in `M2`.

Type resolution of the variable `client` requires sophisticated type inference coupled with a domain-aware preference for finding Boto3 clients wherever they *might* arise and be used for API interaction. In this paper, we present a technique that combines Pyright's type inference with a custom interprocedural dataflow analysis to infer types in such cases.

Furthermore, these API names are exactly the same in Google's Pub/Sub cloud service [19] and AWS's SNS service. Our study shows that the names of some cloud service APIs are exactly the same for cloud services from different commercial cloud vendors (AWS, Google, Tencent, etc.). Thus, precise resolution of service clients' types is extremely important for static analysis of Python applications that use these cloud SDKs.

5 Program Representation

Our analysis represents each program as a collection of per-function graphs called *MU graphs*.[1] A MU graph roughly corresponds to a data-dependence graph overlaid with a control-flow (not control-dependence) graph (CFG). As in prior work that used similar representations [2, 3], we find this representation useful for finding API misuse defects where both the data flowing into an operation and the order of operations are important.

[1] "MU" originally stood for "misuse", and is pronounced as the name of the Greek letter μ.

5.1 MU-Graph Nodes

MU graphs contain five kinds of nodes. **Entry nodes** represent the start of a function's execution: one per MU graph. **Exit nodes** represent the end of a function's execution: one per MU graph. **Control nodes** represent branched control flow, such as a conditional statement or loop. **Action nodes** represent individual execution steps, such as multiplying two values or calling a function. **Data nodes** represent local variables or synthetic temporary values within compound expressions.

Per-node metadata identifies specific uses of these general categories. For example, we distinguish a multiplication action from a function-call action, or an **if**-statement control node from a **while**-statement control node.

Multiple assignments to the same local variable use multiple data nodes, as in static single assignment (SSA) form. ϕ action nodes are added as needed to represent converging data flows, such as when both branches of an **if** statement modify the same variable.

5.2 MU-Graph Edges

Control edges order execution among entry, exit, control, and action nodes. No data node is ever the source or target of a control edge. Thus, discarding all data nodes and non-control edges would reduce a MU graph to a traditional CFG. **Data edges** represent movement of data among control and action nodes, and are further categorized as follows:

Condition edges flow from a data node into a control node, representing the information used to decide how execution continues. For example, a condition edge flows from the value of an **if** statement's predicate to the control node for the statement itself. **Definition edges** flow from an action to a data node defined by that action. For example, a definition edge from a multiplication action to a data node d indicates that d receives the result of that multiplication. **Parameter edges** flow from a data node into an action node. For example, a binary multiplication action is the target of two parameter edges, one for each operand. A function call action is the target of one parameter edge for each actual argument. **Receiver edges** flow from a data node into a method-calling action node. These highlight the special role of implicit self or **this** arguments. **Callee edges** flow from a data node into a call action node, identifying the function to be called. For example, in handlers[event](), an indexing action to fetch handlers[event] would define some temporary data node holding the function to call. A callee edge would then flow from that data node to the call action.

Edges carry additional role-specific metadata. For example, the two control edges that depart from an **if** statement's control node are marked to distinguish the true and false branches. Multiple parameter edges leading to the same action node are ordered, thereby distinguishing an action's first parameter from its second, third, and so on.

5.3 Overall Properties

In the MU representation, data can only flow from data nodes to control/action nodes, and vice versa. Data edges never connect pairs of data nodes directly. Informally, each action node receives zero or more data nodes as inputs, and may provide an output that flows across a definition edge into some other data node. In $x + y * z$, the multiplication action defines an anonymous data node, which in turn flows into the addition action as a parameter.

Figure 3 illustrates several MU-graph features in the representation of $x *= x - 1$, or equivalently $x = x * (x - 1)$. Solid control edges establish evaluation order as in a CFG: subtraction before multiplication, each represented as a rectangular action node. Elliptic data nodes represent two versions of x: x_1 before the assignment and x_2 after. Additional

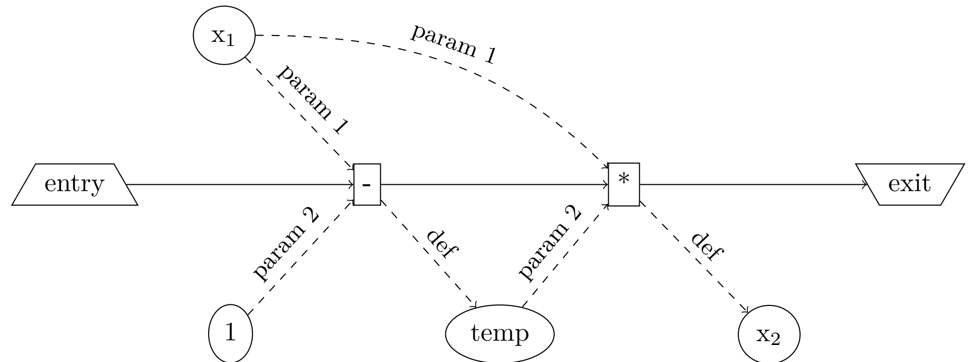


Figure 3 The MU-graph representation of x *= x − 1. Entry and exit nodes are trapezoidal; action nodes are rectangular; data nodes are elliptic. Control edges are solid; data edges are dashed.

data nodes represent the literal 1 and a temporary value. The initial value x_1 is a parameter to both mathematical operations, and is distinct from the final value x_2. The "temp" data node is defined by the subtraction and is also a parameter to the multiplication. Notice that data and non-data nodes strictly alternate along data paths: data nodes provide inputs to action or control nodes, and action nodes' outputs define data nodes.

5.4 Using Pyright for Best-Effort Graph Construction

Pyright is "a fast type checker meant for large Python source bases." [26] Pyright is primarily used behind-the-scenes by Python IDEs, or as a command-line linter/checker. However, Pyright's sophisticated type inference and robust handling of incomplete or incorrect programs make it ideally suited for our purposes as well. MU graph construction begins with a parsed abstract syntax tree (AST) provided by Pyright. We traverse the AST, synthesizing and combining MU graph fragments in a roughly bottom-up manner.

For data nodes, we rely on Pyright to provide static type information and name resolution. Given Python's dynamic nature, these are both best-effort. Inferred static types can be imprecise, absent, or wrong; names can be aliased or accessed covertly via reflection. We attempt no alias analysis or points-to analysis beyond that implicitly performed by Pyright itself. Pyright's best-effort types are available on data nodes that represent named variables as well as those that represent intermediate values, such as the "temp" node in Figure 3.

We flatten data node types to their string representations, such as "int" or "MyClass" or "(int, str) −> tuple[int, str]". Stringification discards internal structure, but allows MU graphs to accommodate essentially any type grammar, even from non-Python languages. Types as strings are also forgiving of incomplete programs: we might know that a piece of data is an instance of MyClass even if we know nothing about MyClass's internal structure or provenance.

The entire process of building MU graphs proceeds, best-effort, even when confronted with imports of missing modules, calls to unknown functions, etc. We represent each questionable operation as some reasonable fallback (e.g., as an empty statement), and move on. Python also contains syntactically ambiguous constructs, such as overloaded operators or the myriad uses of ".". We disambiguate these using types whenever possible, or heuristics when necessary. These approximations mean that we are neither sound nor complete in general. However, these same approximations allow us to provide a representation that is useful in practice when absolute guarantees are not required.

```
CustomRule rule = new CustomRule.Builder()
  .withName("MathExp")
  .withComment("For small floats `x`, the subtraction in `exp(x) − 1` can result in a loss of precision.")
  .withAllOf(
    b −> b.withMethodCallFilter(".*math\\.exp").withDefinitionTransform().as("MathExpResult"),
    b −> b.withConstantDataFilter("1").as("ConstantOne")
  )
  .check()
  .withActionFilter("\\−")
  .withDirectDataFromIdFilter("MathExpResult")
  .withDirectDataFromIdFilter("ConstantOne")
  .build();
```

Figure 4 GQL rule for identifying suboptimal use of the math.exp function.

5.5 From Functions to Programs

The construction process described in Section 5.4 yields one MU graph for each named (**def**) or anonymous (**lambda**) function. For each script, also create MU graph that represents execution of that script's top-level statements.

We aggregate these per-function MU graphs to reflect static program structure. Each Python class contains a dictionary of named methods; each script contains a dictionary of named top-level classes and functions; and so on. We do not build a static call graph, since not all downstream consumers of MU graphs require one. However, we organize and manage the MU graph collection in such a way as to facilitate callee resolution later, if needed.

6 Query Language

Working directly atop the MU representation in authoring analysis rules misses important reuse opportunities. We have therefore designed and implemented an API, dubbed the Guru Query Language (GQL), to enable encapsulation, optimization and reuse of a wide variety of analysis constructs. GQL is implemented as a Java library whose main interface with the analysis builder is the CustomRule class. CustomRule instances are created using the fluent builder pattern [18], where builder calls correspond to reasoning steps in the rule. A rule object can be evaluated at different scopes, from entire code bases to single functions. This is an important source of flexibility, which owes to the MU representation and its support for partial programs. (See Section 5.4.) Rule evaluation yields a RuleEvaluationResult for every type and function that the rule visits, which includes rich information on whether, and if relevant where and how, rule evaluation has failed.

As an illustration of GQL syntax, we refer the reader to Figure 4, where a rule that identifies suboptimal use of the math.exp function is shown. Here is a simple example of what the rule checks for:

```
def foo():
  import math
  return math.exp(1e−10) − 1
```

Rule definition begins by setting the rule's name and user-facing comment text. The following steps, up to the check statement, are preconditions that the rule checks for. Specifically, the withAllOf statement ensures that all the subrules nested within it evaluate successfully, where these check for math.exp calls as well as the presence of the constant value 1. The matches are stored into variables (or IDs), to enable downstream reuse thereof, using the as operation. The actual check, or postcondition, is the rule section after the check step.

It establishes whether there is a subtraction operation that the node defined by `math.exp`, along with the constant `1`, flow into directly (that is, without the mediation of any other action).

6.1 Rule Evaluation

In what follows, we use standard notation, $G = (V, E)$, when referring to MU graphs. Unless stated otherwise, the graphs we mention are specifically MU graphs.

As illustrated above, a GQL rule is an implication relation, $pre \implies post$. As such, rule evaluation is satisfied either when *pre* is not satisfied or when both *pre* and *post* are satisfied. *pre* and *post* are both sequences $[op]$ of operations.

An operation $op\colon \mathbb{P}(\mathcal{V}) \mapsto \mathbb{P}(\mathcal{V})$ is a function whose domain and codomain are both node sets: $\mathcal{V} = \{n\colon \exists G = (V, E).\ n \in V\}$. As an example, a filter operation that matches against calls to a function named "`foo`" evaluates to `foo` call nodes within the incoming node set, if any, or else $\emptyset$.

Given node n, let G_n denote the graph containing n, and $G_n.V$ the complete set of nodes that G_n contains. Operations op satisfy the following two invariants:

1. $\forall N \subseteq \mathcal{V}.\ op(N) \subseteq \bigcup_{n \in N} G_n.V$. That is, application of an operation to a node set N cannot "exceed" the set of nodes due to the graphs containing the nodes in N.
2. $op(\emptyset) = \emptyset$. That is, application of an operation to the empty node set yields the empty node set.

Given rule $r = [op_1, \ldots, op_k] \implies [op_{k+1}, \ldots, op_n]$ and input graph $G = (V, E)$, we denote the node set flowing into op_j as σ_{j-1}. The node sets are defined as follows:

$$\sigma_i = \begin{cases} V & \text{if } i = 0 \\ \emptyset & \text{if } i = k \wedge op_k(\sigma_{k-1}) = \emptyset \\ V & \text{if } i = k \wedge op_k(\sigma_{k-1}) \neq \emptyset \\ op_i(\sigma_{i-1}) & \text{otherwise} \end{cases}$$

Per the first case, precondition evaluation starts from the complete set of graph nodes (V). Per the second and third cases, the transition from precondition to postcondition is either trivial if the precondition has not been satisfied (second case), or – analogously to precondition evaluation – postcondition evaluation starts from V (third case). Any other transition along the operation sequence is simply an application of the operation to its incoming node set.

Rule evaluation is successful if and only if (i) a prefix of *pre* evaluates to $\emptyset$ (in which case the precondition is not satisfied); or (ii) both *pre* and *post* evaluate to non-empty node sets (in which case the precondition and postcondition are both satisfied).

To add color to the formal description so far, rule evaluation is essentially a process of matching against a pattern, or semantic property, where a non-empty node set is a *match frontier* that feeds into the next reasoning step. Failure to maintain a non-empty match frontier means that the given (pre or post) condition is not satisfied by the input function.

6.2 Rule Structure

While our formal presentation above of GQL rules is as logical implication relationships, in practice a rule object has additional information and structure. A GQL rule consists of four sections, as follows: (i) *setup*: the rule's name, and the comment (or description)

associated with the rule; (ii) *function matcher*: a rule can optionally define criteria when to be evaluated, for example based on function name, attributes, annotations, containing type, parameter types, and so on; (iii) *precondition*: the sequence of operations up to the check builder step; and (iv) *postcondition*: the sequence of operations following the check builder step.

Since GQL rules follow the fluent builder pattern, there is risk that users would miss, misuse, or misorder rule constructs or sections. For example, the user might build a rule lacking a check step; forget to set the rule's name; or try to apply incompatible filters in succession. To ensure rule integrity, we employ a hybrid solution that combines metadata contributed by operations with runtime checking. Operations expose a "signature", as explained in Section 6.4, such that improper compositions can be detected and localized ahead of rule evaluation.

6.3 Language- and Domain-specific Rule Constructs

Beyond the core GQL constructs, which are applicable across different programming languages and problem domains, there are reusable albeit language- or domain-specific constructs. As an example, constructs like withNamedArgumentsTransform or withUnpackedArgumentsTransform are useful for Python rules, but do not apply to Java. GQL enables such constructs to be organized into subclasses of CustomRule, such as PythonCustomRule, while containing CustomRule to the core analysis constructs.

This approach has several important advantages. First, we avoid API bloat by distributing analysis constructs across more than just CustomRule. Second, we avoid misuse errors due to a construct being used outside its intended context, for example a Python analysis construct used in a rule that targets Java programs. Finally, GQL extensions sometimes introduce dependencies. We have implemented, for example, a CustomRule extension in the domain of data leaks, where some of the analysis constructs rely on an ML model to predict whether a given data access is retrieving sensitive information. These dependencies should not be forced on GQL users outside the given domain.

6.4 GQL Operations

We now take a closer look at the different operations that comprise GQL rules. These divide into 4 categories, discussed below in turn. Beyond the information in this section, we refer the reader to the accompanying technical report for a more detailed description of the operation categories as well as examples from each category [28].

For safety and fault localization, GQL requires that operations be annotated with their *signature*, which states the types of nodes that they accept as input and yield as output. (See Section 5.) The withReceiverTransform operation, for example, accepts as input action (and more specifically, call) nodes, and outputs data nodes. If a user attempts to compose operations incorrectly, for example by routing the output of a withDataByNameFilter operation to withReceiverTransform, then GQL identifies the violation at runtime and generates a meaningful failure message that localizes it and explains why rule evaluation has been terminated. We are currently in the process of shifting the failure left to rule building time, and as a longer-term objective, compile time.

6.4.1 Core Operations

Core operations apply to all rules, regardless of their scope and logic. Some of the core operations, in particular check and as, have already been explained in the context of Figure 4. Additional core operations include the ability to reset the match frontier, interleave instrumentation (for example, for debugging or profiling purposes), read and write mutable auxiliary state, and so on.

6.4.2 Filter Operations

A filter operation f satisfies the invariant: $\forall V \in \mathcal{V}.\ f(V) \subseteq V$. That is, a filter operation selects a subset of the input node set. Its result cannot exceed the incoming set.

GQL offers a wide selection of built-in filters. Beyond withActionFilter, withMethodCallFilter, withConstantDataFilter and withDirectDataFromIdFilter that are used in Figure 4, there are filters for matching against control structures, constants, actions with specific arguments (like constants or null/None), and so on.

The GQL filter operations – almost without exception – are defined using a unary predicate ranging over nodes, and as such, filtering is done point-wise. As an example, withMethodCallFilter is instantiated through a predicate that accepts action (and specifically, call) nodes where the callee matches the provided regex specification. A common practice with filter operations is to compose them, which enables refinement in pattern matching. An example of that is the consecutive withDirectDataFromIdFilter operations in the rule in Figure 4.

6.4.3 Transform Operations

Transform operations enable the transition from a given match frontier to another frontier that derives from it. For example, a frontier that consists of function calls can be transformed to the respective arguments or receivers, or the values defined by the calls, as illustrated with withDefinitionTransform in Figure 4.

GQL offers many built-in transform operations. Examples include withArgumentsTransform, which transforms an action node to its respective arguments; withControlDependenciesTransform, which transforms a node to its set of control dependencies; withDataDependenciesTransform (*resp.* withDataDependentsTransform), which transforms a node to its set of (transitive) data dependencies (*resp.* dependents); and withReceiverTransform, which transforms a call node to the receiver (if available).

6.4.4 Second-order Operations

Logical structures and operators are necessary to express certain rule logic in a precise and concise manner. As a simple example, the user may wish to check if a given function call "zoo" has a receiver of type either Foo or Bar. Another use case, illustrated in Figure 4, is the need to check that several conditions are all met through withAllOf.

To enable such control and logical structures, GQL exposes second-order operations. These are operations that are themselves parameterized by one or more rules, which we refer to as *subrules*.

As an illustration, here is the GQL syntax for the above example:

```
.withMethodCallFilter("zoo")
.withOneOf(
  b -> b.withReceiverByTypeFilter("Foo"),
  b -> b.withReceiverByTypeFilter("Bar"))
```

The withOneOf construct evaluates to the first subrule that yields a non-empty result, or else it evaluates to $\emptyset$.

6.5 Interprocedural Analysis

As noted above, GQL provides the ability to perform interprocedural analysis through the withInterproceduralMatch construct and several specializations thereof. The underlying call-graph representation resolves call sites on demand, per the CHA call-graph construction algorithm [21], based on the (i) name of the callee, (ii) number of arguments, and (iii) argument types. Though the CHA algorithm is known to be imprecise [11], we have rarely seen cases where that was the cause of imprecision in GQL rule evaluation. We hypothesize that this is because (i) interprocedural analysis is run at file or package scope, but not beyond, so there is less room for error, plus (ii) imprecision in interprocedural analysis is potentially mitigated by other rule steps.

At a high level, the interprocedural tracking algorithm performs a fixpoint computation starting from the seeding function graph and matched nodes therein. At each step, the argument rule is applied to match against additional nodes. The algorithm is parametric, enabling the user to decide the scope (intra-class, intra-file, or entire codebase) and direction (forward or backward) for tracking. In the forward direction, the algorithm transitions from a call site to the callees and from a function's exit to callers. In the backward direction, the algorithm transitions from a function's entry to call sites and from call-site definitions (for example, x = foo(), where x is tracked) to callee exits.

Functional summaries are utilized to avoid redundant computation. In the forward direction, these document the relationship between a call-site argument and the definition (if exists) plus other arguments. In the backward direction, the summary documents the relationship between the definition and call-site arguments.

A more complete, and technical, explanation of the GQL interprocedural tracking algorithm is available in the accompanying technical report [28]. The description there ties into a pseudocode description of the algorithm.

6.6 Dataflow Analysis

Beyond its interprocedural capabilities, GQL also has built-in support for several flavors of dataflow analysis, including slicing and taint tracking.[2] These build directly on top of the data edges exposed by the MU representation, in conjunction with the interprocedural matching algorithm described above.

The main feature that the GQL dataflow analysis provides beyond a standard fixpoint algorithm over the dataflow relation is the ability to specify matchers on graph edges to tag them with unique roles: *passthrough* (data flows across the call site), *blocking* (an edge being either a sanitizer or a validator), *side effecting* (data flows into the receiver of a call), or *reading* (data flows from the receiver to the definition). The user-provided specification is then enforced as part of the fixpoint algorithm.

7 Type Inference for Boto3 Clients

As explained in Section 3.1, a Python AWS application creates an AWS service client by passing the name of the service as an argument to one of two distinct levels of APIs. The use of these multiple API flavors, the interactions between them, and the use of strings as service selectors, all pose challenges for type inference.

[2] GQL additionally features finite state machine (FSM) and typestate analysis, though these involve not just dataflow but also control-flow reasoning. These capabilities are not consumed by the rules that we discuss later in the paper, so we suffice by noting them here.

Regardless of which API is used, AWS service clients are ultimately just data values. Like any other data, service clients can be stored in class variables, assigned into global variables, returned from functions, and so on. Code might use a service client locally within a single function or globally within or even across the files that comprise the complete application. The complexity of these *definition–use chains* (DU chains) further complicates type inference.

In this section, we present different type inference strategies that can be used in this challenging application domain.

7.1 Pyright's Type Inference With Boto3-Stubs

Pyright supports type inference for function return values, instance variables, class variables, local variables, and global variables. Pyright's inference engine uses several advanced type inference techniques, such as a flexible model of "type assignability", inferred types for self and cls, parameterized generic types, including both polymorphic container types as well as optional types, union types representing arbitrary sets of possible types, overloaded function types as a special case of union types for *ad hoc* polymorphic functions, literal types, such Literal["str"] as a subtype of **str** that represents only the string literal "s3", and few others.

A full discussion of these capabilities is outside of the scope of this paper, and in any case Pyright is not our contribution. We treat Pyright's type inference engine as a powerful, featureful, but opaque black box.

If Pyright cannot infer the type of some symbol, then that symbol's type is set to Any. This fallback type is a useful warning marker that lets inference consumers (such as CodeGuru) recognize cases where Pyright type inference fell short.

Type inference can incur significant computation overhead for large code bases. Also, Pyright cannot always infer correct types without some outside help. Hence, type annotations are a practical requirement for building a robust type inference system. We use third-party type stubs, called *Boto3-stubs* [14], that provide full type annotations for Boto3. Pyright ingests type annotations provided by Boto3-stubs to further enhance and constrain its type inference.

Figure 2 give an examples of Pyright's Type Inference with Boto3-stubs (denoted by the prefix "#→"). However, in Figure 7, Pyright fails to infer a precise type for s3_client in the method load_df_from_s3, instead giving it the fallback Any type.

7.2 Type Inference Using Custom Dataflow Rules

As an alternative to Pyright, we have used GQL to implement custom inference rules based on dataflow analysis. These rules do not provide universal, generic type inference. Instead, they focus on idiomatic, interprocedural Boto3 usage patterns that Pyright's general-purpose engine fails to address. There are a total of ten GQL-based custom dataflow rules, among which only one is intraprocedural rule and rest nine are interprocedural rules. For illustration purpose, we select few representative interprocedural GQL rules that have low to medium complexity (in terms of number of operations in the rules) and that performs dataflow analysis at file-scope or package-scope.

7.2.1 Representative Examples of Interprocedural Rules

Each GQL rule in Figures 5–6 implements some form of interprocedural dataflow analysis. Each operates on a function graph and matching API nodes along with the receiver nodes of calls to the corresponding APIs. For example, in Figure 7, one relevant API node is

```
builder −> builder
  .withInterproceduralMatch(
    new InterproceduralMatchOperation.InterproceduralMatchSpec(
      /* scope = */ InterproceduralMatchOperation.Scope.FILE_FORWARD_REACHABLE,
      /* stopOnFirstMatch = */ false,
      /* visitAllNodes = */ false),
    bb −> bb.withDataDependentsTransform(
      /* isTransitive = */ true,
      /* isInterprocedural = */ true))
  .withOneOf(
    bc −> getBoto3Client(bc, service)
  )
```

Figure 5 Rule example using forward, interprocedural dataflow.

```
builder −> builder
  .withInterproceduralMatch(
    new InterproceduralMatchOperation.InterproceduralMatchSpec(
    /* scope = */ InterproceduralMatchOperation.Scope.FILE_BACKWARD_REACHABLE,
    /* stopOnFirstMatch = */ false, /* visitAllNodes = */ false),
    bb −> bb.withDataDependenciesTransform(
      /* isTransitive = */ true, /* isInterprocedural = */ true))
  .withOneOf(bc −> getBoto3Client(bc, service))
```

Figure 6 Rule example using backward, interprocedural dataflow.

get_object, for which the corresponding receiver node is s3_client. Our strategy for resolving call actions to callees is name-based: we match the name of the API entry point (callee) in the code against API specifications that are extracted from Boto3.

Figure 5 shows one such rule. The scope of this rule's interprocedural match operation is FILE_FORWARD_REACHABLE, which directs GQL to track dataflow forward using a "data dependents" transform operation that transforms from incoming nodes to nodes that are data dependent on them, including in other functions. The result of this interprocedural tracking is then checked to determine if it matches one of the known flavors of Boto3 clients (low-level or object-oriented), by calling the utility methods inside the withOneOf operation.

The rule in Figure 6 implements interprocedural backward dataflow analysis, complementary to the forward analysis of Figure 5. For the backward version, tracking is specified as FILE_BACKWARD_REACHABLE. This scope directs the interprocedural analysis to perform backward dataflow tracking using a "data dependencies" transformer that transforms from incoming nodes to nodes that are data dependent on them, including in other functions. Similar to the previous rule, this rule's withOneOf clause then checks whether the result of backward interprocedural tracking matches one of the known flavors of Boto3 clients.

7.2.2 Example of Type Inference Using Custom Dataflow Rules

Figure 7 shows a Python code snippet with variable- and function-level type annotations from Pyright. The type of s3_client in the method write_df_to_s3_location is correctly inferred as S3Client: an Amazon S3 service client. This client is passed via input parameter to the method load_df_from_s3. In absence of the type annotation for the input parameter, Pyright could not infer the type of s3_client (denoted by Any), inside the method load_df_from_s3.

However, one of our custom dataflow rules can resolve the type of s3_client in method load_df_from_s3. The applicable rule starts from a matching API node, s3_client.get_object, where the type of the receiver node s3_client needs to be determined. Recall that the matching

```
def write_df_to_s3_location(file_path, bucket_name, metadata, sep=None):
  s3_client = create_s3_client()
  #→ s3_client: S3Client
  load_df_from_s3(s3_client, bucket=bucket_name, path="")
  s3_client.put_object(Body=file_path, Bucket=bucket_name)

def create_s3_client():
  return Boto3.client("s3")
  #→ create_s3_client: () -> S3Client

def load_df_from_s3(s3_client, bucket, path):
  raw_data = s3_client.get_object(Bucket=bucket, Key=object_path)
  #→ s3_client: Any
```

Figure 7 Type annotation for AWS client passed by input parameter.

API node is obtained by matching the name of the API against the API specification extracted from Boto3. Starting from a matching API node, the rule uses a "parameter transform" operation that transforms incoming nodes to the parameters of the respective functions. This rule then uses a "backward data dependencies" transform that transforms from incoming nodes to their data dependencies, including in other functions. The rule's result includes the node `s3_client` in the method `write_df_to_s3_location`, whose type is already known to be `S3Client`. It is worth noting that the type of `s3_client` could also be inferred by a stand-alone custom dataflow rule (in absence of type annotations from Pyright). However, the rule specification would be more complex. We prefer to augment Pyright's capabilities rather than replace them.

7.3 Layered Type Inference

The example in Figure 7 shows that a hybrid approach for type inference can combine custom dataflow rules with Pyright's type inference to resolve types that Pyright cannot resolve by itself. Each of these type inference approaches have complementary strengths. This quality suggests a layered approach for type inference that combines these strategies in a staged manner. Our layered approach first uses Pyright's type inference with Boto3 stubs to infer type annotations for at least some Boto3 clients. Per Section 5.4, data nodes in MU graphs carry type metadata reflecting Pyright's inference results. If the type of an API call of interest is already known, then that may be sufficient to recognize that the API belongs to Boto3. If the type of the API call of interest is unknown, then our layered approach deploys custom dataflow rules to infer client types. Section 9 presents our empirical evaluation of the strengths and limitations of this layered approach.

8 AWS Best Practices Rules

In this section, we describe a representative sample of eight rules that detect different types of defects related to usage of the Boto3 API. These rules cover approximately 200 public-facing AWS services. All Python AWS best practices rules (as well as most other CodeGuru rules) are implemented atop GQL (see Section 6), and follow the same rule evaluation mechanism that is discussed in Figure 4. Of the eight rules discussed in this section, we focus in particular on two rules – concerning pagination and batchable APIs – to enable thorough discussion of rule syntax and sample detections.

```
def sync_ddb_table(source_ddb, destination_ddb):
    response = source_ddb.scan(TableName="table1")
    for item in response['Items']:
        destination_ddb.put_item(TableName="table2", Item=item)
```

Figure 8 Non-compliant Pagination Example.

```
def sync_ddb_table(source_ddb, destination_ddb):
    response = source_ddb.scan(TableName=="table1")
    for item in response['Items']:
        destination_ddb.put_item(TableName="table2", Item=item)
    # Keeps scanning until LastEvaluatedKey is null
    while "LastEvaluatedKey" in response:
        response = source_ddb.scan(TableName="table1",
                                   ExclusiveStartKey=response["LastEvaluatedKey"])
        for item in response['Items']:
            destination_ddb.put_item(TableName="table2", Item=item)
```

Figure 9 Correct Pagination Example.

Worthy of mention is our ability, thanks to the AWS best practices rules and their detections, to form an effective collaboration between the CodeGuru and AWS SDK teams. From our side, the collaboration consists of frequent feedback to the SDK team (either conveying developer feedback or trends that we observe across multiple detections). From the AWS SDK team's side, our rules and detection technologies pose as a platform to promote awareness of new features, for example the SDK V2 pagination feature.

8.1 Detecting Misuse of Paginated APIs

The pagination trait is implemented by over 1,000 APIs belonging to >150 AWS services. This trait is commonly used when the result set due to a query is too large to fit within a single response. For the complete set of results, a pagination token is used to perform iterative requests and receive the response in parts. Developers who are not aware of this trait might mistakenly suffice with a single request/response result, as illustrated in Figure 8.

Here the `scan` call is used to read items from an Amazon DynamoDB table, where `put_item` saves those items to another DynamoDB table. The `scan` API implements the pagination trait. However, the code neglects to check for additional results beyond the initial batch, which is clearly wrong. Our pagination rule detects the missing pagination in this example, and generates a recommendation to iterate on the complete result set through the `LastEvaluatedKey` token available through `response`. A compliant version of the code, consistent with this recommendation is shown in Figure 9.

8.2 Error Handling for Batch Operations

More than 20 AWS services expose batch APIs, which enable bulk request processing. Batch operations can succeed without throwing an exception even if processing fails for some items. Therefore, a recommended best practice is to explicitly check for failures in the response due to the batch API call. We illustrate incorrect and correct usages of batch APIs in Figures 10 and 11, respectively.

The rule for detection of batch operations where failures are not checked is shown in Figure 12. Like many other CodeGuru rules, in particular in the AWS best practices category, this rule is parameterized by a configuration. (See Section 9 for an example.)

```
def noncompliant():
  sqs = boto3.client('sqs', 'us-west-2')
  sqs.send_message_batch()
```

Figure 10 Incorrect Error handling for Batch Operation example.

```
def compliant():
  sqs = boto3.client('sqs', 'us-west-2')
  response = sqs.send_message_batch()
  if "Failed" in response:
    raise SendMessageToSQSFailure("Failed")
```

Figure 11 Correct Error handling for Batch Operation example.

The rule's precondition searches for batch API calls per the configuration, then transforms from the calls to their respective receivers, which are stored into variable AWS_CLIENT. Backward propagation, in an attempt to relate these receiver nodes to applicable Boto3 services, then takes place through the getBoto3 call.

The postcondition loads the batch API call, stored as variable BATCH_API_CALL, then checks whether the result of the call is ignored through withOutputIgnoredFilter. This filter checks whether the call node(s) flowing into it define(s) a node that has no outgoing edges.

8.3 Other Representative Rules

We now switch to additional rules in the AWS best practices category, and provide an explanation of what they each check for.

Use waiters in place of polling API: Waiters are utility methods that make it easy to wait for a resource to transition into a desired state by abstracting out the polling logic into a simple API call. The waiters interface provides a custom delay strategy to control the sleep time between retries, as well as a custom condition on whether polling of a resource should be retried. Our rules detect code that appears to be waiting for a resource before it runs. In such cases, it recommends using the waiters feature to help improve efficiency.

Detect missing None check on cached response metadata: Response metadata represents additional information included with a response from AWS. Response metadata varies by service, but all services return an AWS request ID that can be used in the event a service

```
PythonCustomRule.Builder()
    .withMethodCallFilter(config.api)
    .as(BATCH_API_CALL)
    .withReceiverTransform()
    .as(AWS_CLIENT)
    .reset()
    .withClosure(
    /* Pre-condition: Match that the type of API is a Boto3 client */
    b -> getBoto3Client((PythonCustomRule) b, serviceId, AWS_CLIENT))
    /* CHECK */
    .check()
    /* Post-condition: Check that the output of Boto3 API is ignored */
    .withId(BATCH_API_CALL)
    .withOutputIgnoredFilter()
    .build();
```

Figure 12 Rule to check for batch API calls sans failure checking.

call isn't working as expected. If the code attempts to access the response metadata, ResponseMetadata, without performing a None check on the response object, then this might cause a NoneType error. To prevent this, our rule recommends adding a None check on the response object before accessing the response metadata.

Detect failed records in Kinesis PutRecords: The put_records operation in AWS Kinesis service might fail, thereby causing loss of records. This rule detects if the code handles the failed records from the put_records operation. In the absence of such handling of failed records, the rule recommends checking the FailedRecordCount in the put_records response to see if there are failed records in the response. A failed record includes ErrorCode and ErrorMessage values. If failed records are found, the rule recommends adding them into the next request.

Detect deprecated APIs: This rule detects usage of deprecated APIs in Python application code. A total of 107 deprecated API specifications are extracted from Boto3, identified from the use of deprecated trait in the API models. These API specifications are fed into the rule for detecting deprecated APIs in real world Python code.

Detect inefficient/redundant API chains: The rule for inefficient/redundant API chains detects usage of less performant APIs or outdated APIs, an API call chain that could be replaced with a single API call, a manual pagination operation where the SDK provide a Paginator API to automatically perform the pagination, and much more.

Detect expensive client object construction in Lambda handler: This rule detects a Boto3 client that is initialized from a Lambda handler. In order to speed up Boto3 client initialization and minimize the operational cost of the Lambda function, the rule recommends creating the client at the level of the module that contains the handler, and then reusing it between invocations. This is stated in the best practices for the lambda handler. [5]

9 Experimental Results

In this section, we report on experiments to validate our approach for on-demand resolution of Python types. Our experiments are guided by the following research hypotheses:

Hypothesis 1: Skipping type inference, instead relying solely on function names and arguments, is insufficient since that might lead to excessively many false positive detections.

Hypothesis 2: The dataflow-based and Pyright-with-stub-based resolution strategies have complementary strengths.

Hypothesis 3: A staged approach that combines dataflow and stubs with name-based resolution as a low-confidence fallback is effective.

Hypothesis 4: The AWS best practices rules, running atop the staged algorithm, are sufficiently precise, efficient and actionable to provide value during code review.

We note that beyond type inference, once a function call is confirmed to invoke a given AWS service, most of the rules are straightforward and do not require complex and/or interprocedural analysis to detect incorrect or suboptimal use of the AWS API. There are few exceptions, where the actual rule's logic can be imprecise, but overall the correctness of type inference is a good proxy for the correctness of a rule finding.

We illustrate rule dependence on identification of the Boto3 service being invoked using the JSON snippet below, taken from our service's production configuration. The "Missing Pagination" rule, whose specification is described in the snippet, searches for paginated functions like list_dataset_groups in the specific context of the forecast AWS service. Recall that these API specifications are automatically extracted from the API models in Boto3.

Table 1 Number of type resolutions due to each of the resolution strategies.

Strategy	Confidence	Description	Type Resolution Count	Precision
1	1.0	Pyright with Boto3 type stubs	2,293	100 %
2	1.0	Dataflow tracking	3,065	100 %
3	0.5	API name based resolution	5,403	54 %

```
{
  "expectedPaginationMethods": [
    "IsTruncated",
    "NextToken"
  ],
  "paginatedMethod": "list_dataset_groups",
  "resultKeys": [
    "DatasetGroups"
  ],
  "serviceId": "forecast"
}
```

We have evaluated the strategies described in Section 7 using a dataset consisting of 3,027 public GitHub repositories. These repositories were selected based on the following criteria: (1) The repository contains Python source files (at least 3, and with a total of at least 100 lines of code). (2) The repository has an MIT or Apache license. (3) The repository has a rating of 3 stars or more. (4) The repository makes use of the AWS SDK.

9.1 Performance of Resolution Strategies in Isolation

To examine the first two hypotheses laid out above, we begin by computing precision and recall for the different type resolution strategies in isolation. Precision is measured as the proportion of correct (TP) versus incorrect (FP) type resolutions, and recall is measured as the proportion of correct (TP) versus missed (FN) type resolutions. In what follows, we use the notation $t[s]$ to refer to the type of SDK service client s.

9.1.1 Type-Resolution Strategies

We consider 3 different strategies for resolution of $t[s]$:

Strategy 1: Use Pyright's type inference in conjunction with third-party Boto3 type stubs. This strategy potentially recover types beyond the boundaries of a single function.

Strategy 2: Use interprocedural dataflow analysis, combining backward and forward queries.

Strategy 3: Match against the API name without attempting to resolve the type of the receiver, which is an over-approximate yet cheap approach.

9.1.2 Results

Table 1 shows the number of resolutions due to each of the strategies when applied to the GitHub dataset. To gain qualitative insight into the results, and how many of the type resolutions are accurate, we manually reviewed 50 Boto3 client detections, selected at random, for each of the three strategies for a total of 150 detections. Reviewers consisted of five senior engineers and scientists, all expert users of the Boto3 library.

Our qualitative analysis suggests that strategies 1 and 2 are highly precise, as reported in the "Precision" column of Table 1. All 50 cases sampled for manual review were judged as correct. By contrast, for strategy 3, only 54% of the samples (27 out of 50) were correct. By definition, strategy 3 achieves 100% recall and thus establishes an upper bound on the number of false negatives due to strategies 1 and 2.

The set of detections obtained from strategy 1 and strategy 2 are not exactly the same, and they do not subsume each other: some strategy-1 detections are omitted by strategy 2, and vice versa. Out of 27 true positive detections from strategy 3, 19 detections are also obtained from strategy 1 and strategy 2 combined. The remaining 8 detections (30%) are exclusive to strategy 3.

9.1.3 Discussion

We consider the pros and cons of the three strategies in light of these results.

Strategy 1 uses third-party Boto3 type stubs, together with Pyright's type inference to resolve AWS SDK clients. Unlike strategy 2, where type resolution occurs *during* rule evaluation, strategy 1's Pyright-derived types are available *before* rule evaluation, during MU graph construction. This allows type resolution to run once rather than on every application of every rule: a major performance boost.

On the negative side, strategy 1 suffers from low recall, as shown in the "Type Resolution Count" column of Table 1. This is due to the different ways in which AWS SDK clients are obtained, and in particular, the common case of passing them as function parameters. Pyright does not search for callers of the function, thus assigning `Any` as the type of the parameters unless annotations are explicitly provided.

Moving to strategy 2, the ability to perform backward dataflow tracking addresses the challenge of passing AWS SDK clients as function parameters. Duplication of work on type resolution is mitigated by a staged algorithm that first attempts intraprocedural resolution, then performs tracking at the file level, and finally at the level of the entire codebase. From our experience, and performance measurements, the staged algorithm is quite effective. Like strategy 1, strategy 2 retains full precision, yet has much higher recall as shown in the "Type Resolution Count" column of Table 1.

In spite of its overall effectiveness, strategy 2 – which tracks dataflow through local variables – can miss cases where the client is stored as a field or global variable. These cases are handled by strategy 1.

Our analysis of the gaps between strategies 1 and 2 is confirmed experimentally. In line with hypothesis 2, we have found 60 detections that are exclusive to strategy 1 and 832 detections that are exclusive to strategy 2.

Finally, the low precision of strategy 3 (just over 50%) confirms hypothesis 1. At the same time, the computational cost of strategy 3 is virtually zero, and thanks to its simplicity, it is able to sometimes completely bypass complex tracking scenarios that are beyond the power of strategies 1 and 2. An example is given in Figure 13, where neither strategy 1 nor strategy 2 is able to recognize that `self._ec2_client` is a Boto3 client in the body of the `ec2_client.describe_snapshots(**kwargs)` method. Strategy 3 succeeds here simply by recognizing `describe_snapshots` as the name of an AWS SDK client API method.

To make use of strategy 3 in spite of its approximate nature, we "penalize" detections due to this strategy by assigning a confidence score of 0.5 to those detections compared to 1.0 if the detection is due to strategies 1 or 2, as shown in the "Confidence" column of Table 1. The exact value of 0.5 is arbitrary, but serves to distinguish the lower-confidence

```
class AwsClient(object):
    def __init__(self, *args, **kwargs):
        self._boto3client = None
        super(AwsClient, self).__init__(*args, **kwargs)

    def create_ec2_client(self, context=None):
        #→ (method) create_ec2_client:
            (self: Self@AwsClient, context=None) -> Any
        return boto3.client('ec2')

    def get_aws_client(self, context):
        if not self._boto3client:
          ec2_client = self.create_ec2_client(context)
          #→ (variable) ec2_client: Any
        return self._boto3client

    def describe_snapshots(self, **kwargs):
        response = self._ec2_client.describe_snapshots(**kwargs)
        #→ (variable) _ec2_client: Any
```

Figure 13 Detections from Strategy 3 that strategies 1 and 2 miss.

detections of strategy 3 from the higher-confidence detections of strategies 1 or 2. This is in line with our earlier comment that the correctness of type resolution is a good proxy for the correctness of a detection.

9.2 Performance of Combined Resolution Strategies

The results in Section 9.1.1 suggest that there is benefit in combining the different strategies in light of their complementary strengths. Starting from this motivation, we report here on experiments with "hybrid" resolution strategies, which we refer to as *configurations*.

9.2.1 Type Resolution Configurations

We consider two configurations: High Confidence (HC) runs strategy 1, then strategy 2 where needed to complement strategy 1. Mixed Confidence (MC) runs strategies 1 and 2 in the same fashion as HC, but rather than giving up if both fail, proceeds to strategy 3 in an attempt to generate a low-confidence detection.

CodeGuru uses the confidence score to rank the detections as per the "Confidence" column in Table 1. Detections from strategy 1 and strategy 2 rank higher than detections from strategy 3 thanks to their higher confidence score. CodeGuru imposes different restrictions and limitations on detectors, in particular with regard to the overall number of detections, which means that in the presence of sufficiently many high-confidence detections, low-confidence detections are suppressed. By implication, low-confidence MC detections are not always reported to the user.

9.2.2 Results

Table 2 reports results for both configurations, running against the dataset of 3,027 GitHub repositories. The total time for running each configuration is close to 5 hours.

In line with hypothesis 2, the HC configuration generates more detections than strategies 1 or 2 in isolation. The total number of detections due to the HC configuration is 60 more than strategy 2: exactly the number of detections that are exclusive to strategy 1.

Table 2 Type Inference Configurations.

Configuration	Strategies	Description	Number of Detections
HC	1, 2	Pyright with stubs followed by dataflow	3,125
MC	1, 2, 3	All layers	5,403

Moving to the MC configuration, the number of detections that it generates is identical to strategy 3 in isolation, which is expected. The important difference, however, is that most (that is, 3,125) of the detections have high confidence, with only 2,278 detections relying on strategy 3.

Projecting from the detections we sampled and triaged, we estimate that the MC configuration has a precision score of 0.85 along with perfect recall, whereas the HC configuration has perfect precision but a recall score of roughly 0.72 (with the assumption that 54% of the findings found by MC but not HC are true positives). This analysis supports hypothesis 3, which favors use of strategy 3 as part of the combined strategy rather than relying only on the high-confidence strategies.

9.3 Real-world Feedback on the Rules

Beyond our offline study, we also report on data from the field driven by comments that CodeGuru has left on code reviews in production. CodeGuru posts comments on code reviews just as a human reviewer would. We have augmented the comment UI with a feedback menu, so that a developer can optionally rate a detection as "Useful", "Not Useful" or "Not Sure" and/or provide free-form textual feedback. These feedback mechanisms give the CodeGuru team insight into the performance of different detectors and enable detector tuning over time.

For AWS best practices, each CodeGuru comment contains two key fields:

1. One or two paragraphs explain what the issue is, and why fixing it is important. For example, in the case of a batch operation whose output is ignored, the explanation states that even if some items are not processed successfully, the batch operation might still complete successfully without raising an exception.
2. A "Learn More" hyperlink directs the user to the appropriate section in the Boto3 online documentation for complete information on the API in question.

We provide lower-bound metrics to give a sense of the size of CodeGuru's input funnel. In the studied time period of 10 weeks, CodeGuru analyzed $\gg 1,000,000$ lines of code. We applied $\gg 10$ detectors, yielding $\gg 10,000$ AWS best practice recommendations, which we reported to $\gg 1,000$ developers.

We note that by definition, the codebases involved in this study are all live (undergoing code reviews and modifications). These are Python cloud services and applications that make use of Boto3, where the developers are industry practitioners with Python and cloud background. Hence we assign high weight to their feedback on CodeGuru detections.

In CodeGuru, we measure *acceptance* as an indication of whether or not developers have found a given rule's review comments useful. Given a set of "Useful" (U), "Not Useful" (NU) and "Not Sure" (NS) ratings, we compute acceptance as the ratio $\frac{|U|}{|U|+|NU|+|NS|}$, where by $|U|$ we mean the number of "Useful" feedback points, and analogously for NU and NS. Note, importantly, that we conservatively treat "Not Sure" the same as "Not Useful".

Table 3 Acceptance rate per rule from developer feedback during code review.

Rule	Acceptance Rate
Detect missing Pagination	75.0 %
Data loss in Batch APIs	100.0 %
Use Waiters instead of Polling APIs	52.0 %
Detect failed Records in Kinesis PutRecords	100.0 %
Detect deprecated APIs	88.9 %
Detect usage of inefficient/redundant API chains	85.7 %
Missing None check on cached response metadata	85.7 %
Detect expensive client object construction in Lambda handler	75.8 %

Table 4 Breakdown of the detections from Table 3 by confidence level.

	Proportion of Detections	
Detection Group	High Confidence	Low Confidence
All	88 %	12 %
Accepted	93 %	7 %
Not Accepted	84 %	16 %

Table 3 shows the acceptance data for eight of the Python AWS best practices rules for a time period of 10 weeks. We obtained $\gg 100$ feedback points from a population of $\gg 100$ developers through the feedback UI described above. As reported in Table 3, developers accepted over 85% of the recommendations made by five out of the eight rules, and almost 83% of the overall recommendations.

Only one of the eight rules, "Use Waiters instead of Polling APIs", has an acceptance rate below 75%. Our analysis of this rule's performance, including communication with some of the developers who left feedback on its detections, suggests that the gap between acceptance and correctness is important. Developers often acknowledge the detection as correct, but push back for one or more of the following reasons: (1) The intent of the PR is different, and they prefer not to merge multiple unrelated changes into the same PR. (2) The change is applicable, but requires upgrading the codebase to use the latest AWS Python SDK, which again exceeds the scope of the PR. (3) The change is not applicable, since the code in question is test code or there is no concern about polling in the given context. It is worth adding that outside the time period reported here, we have seen multiple weeks where acceptance rate for "Use Waiters instead of Polling APIs" was high.

Overall, acceptance data from the field supports hypothesis 3 in showing that developers mostly find the detections by to the Python AWS best practices rules useful. These are made using the MC configuration, which integrates all three of the resolution strategies described in Section 9.1.1.

From our conversations with developers, the textual feedback they provided, and our own review of some of the detections and their corresponding feedback, we have identified two main factors that contribute to the usefulness of our rules: (1) Missed features: SDK changes across versions, in particular new features, are sometimes missed by developers. Pagination, retry and error handling are examples of such features, where developers not familiar with these built-in capabilities sometimes implement "manual" mechanisms instead. Another

example is manual polling versus the recommended use of the waiter utility. (2) Missed expectations: Developers sometimes assume, rather than verify, the functionality of a given API or the role of a given parameter. An example is the QueryResponse::hasItems method, whose (boolean) return value is sometimes incorrectly interpreted to mean that the response contains a non-empty collection of items, where what is in fact meant is that response defines an Items property. To make sure whether any items are contained in the response, the developer needs to also check Items::isEmpty. Mistakes like this can lead to large-scale operational failures.

Table 4 reports the breakdown, by confidence level (high versus low), for the detections in Table 3. In sharp contrast to the distribution due to strategy 3 from the offline study, where approximately 45% of the detections had a low confidence score, the hybrid inference strategy leans heavily towards high-confidence detections (88% of all detections). This is consistent with the suppression policy described above, in Section 9.2.1, for low-confidence detections. The tradeoff that the hybrid strategy offers in the presence of confidence-based suppression is appealing, in that low-confidence detections are typically shadowed by high-confidence detections, which limits the impact of such detections on precision and allows them to play an important role in pushing coverage upwards when high-confidence detections are absent. Also note, from Table 4, that the proportion of low-confidence detections among "Not Accepted" detections is higher compared to "Accepted" detections (16% versus 7%), which is consistent with the data from the offline study.

Overall, our analysis of detections from the field, and how these map back to the hybrid strategy, are in support of hypothesis 4. Developers tend to view our AWS best practices recommendations as useful. Most of the recommendations build on high-confidence type inference, with some remaining cases benefiting from the low-confidence resolution strategy.

10 Conclusion and Future Work

We have presented an industrial-strength framework for precise static analysis of Python applications that use AWS cloud services. In support of this goal, we have developed a novel type inference system for identifying and tracking AWS service clients in real-world Python applications. Our Python MU graph IR is suitable for building a wide range of static analyses or best-practice rules for Python applications. Furthermore, the Guru Query Language provides the right level of abstraction with its encapsulation, optimization and reuse features to develop static analysis rules that can be evaluated at different scopes, from single functions to entire applications.

Experiments on 3,027 open-source Python GitHub repositories show that individual inference strategies have complementary strengths. The most effective solution, then, is a layered approach that combines Pyright with Boto3 stubs, custom dataflow analysis in GQL, and name-based resolution as a low-confidence fallback. Our layered strategy achieves 85% precision and 100% recall in typing relevant Boto3 values in Python client code. The ultimate authorities on the value of our approach are real-world developers, with no ties to the authors. Those developers accepted more than 85% of the recommendations made by five out of eight rules, and roughly 83% of the recommendations on average.

In the future, we plan to extend and generalize our type inference infrastructure to other rule suites and properties that apply to Python programs. We are also examining ways to reuse our work on Python on-demand type inference when adding support for other dynamically typed languages.

References

1 Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

2 Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. Investigating next steps in static API-misuse detection. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 265–275. IEEE / ACM, 2019. doi:10.1109/MSR.2019.00053.

3 Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. A systematic evaluation of static API-misuse detectors. *IEEE Trans. Software Eng.*, 45(12):1170–1188, 2019. doi:10.1109/TSE.2018.2827384.

4 Amazon Web Services. AWS SDK for Python (Boto3) [online]. URL: https://aws.amazon.com/sdk-for-python/ [cited 2022-05-12].

5 Amazon Web Services. Best practices for working with AWS Lambda functions: Function code [online]. URL: https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html#function-code [cited 2022-05-12].

6 Amazon Web Services. Boto3 - the AWS SDK for Python [online]. URL: https://github.com/boto/boto3 [cited 2022-05-12].

7 Amazon Web Services. Boto3 developer guide: Low-level clients [online]. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/clients.html [cited 2022-05-12].

8 Amazon Web Services. Boto3 developer guide: Resources [online]. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/resources.html [cited 2022-05-12].

9 Amazon Web Services. What is Amazon CodeGuru Reviewer? [online]. URL: https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html [cited 2022-05-12].

10 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In Pascal Costanza and Robert Hirschfeld, editors, *Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, October 22, 2007, Montreal, Quebec, Canada*, pages 53–64. ACM, 2007. doi:10.1145/1297081.1297091.

11 David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Lougie Anderson and James Coplien, editors, *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996*, pages 324–341. ACM, 1996. doi:10.1145/236337.236371.

12 Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. PYInfer: Deep learning semantic type inference for Python variables. *CoRR*, abs/2106.14316, 2021. arXiv:2106.14316.

13 Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna M. Reinen. Ariadne: analysis for machine learning programs. In Justin Gottschlich and Alvin Cheung, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 1–10. ACM, 2018. doi:10.1145/3211346.3211349.

14 Vlad Emelianov. mypy_boto3_builder: Type annotations builder for boto3 compatible with VSCode, PyCharm, Emacs, Sublime Text, pyright and mypy [online]. URL: https://vemel.github.io/mypy_boto3_builder/ [cited 2021-12-01].

15 Facebook. Pyre [online]. URL: https://pyre-check.org/ [cited 2021-11-30].

16 Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for Python. In Ulrik Pagh Schultz and Jeremy Yallop, editors, *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, Paris, France, January 18-20, 2017*, pages 89–98. ACM, 2017. `doi:10.1145/3018882.3018888`.

17 Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static value analysis of Python programs by abstract interpretation. In Aaron Dutle, César A. Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 185–202. Springer, 2018. `doi:10.1007/978-3-319-77935-5_14`.

18 Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993. `doi:10.1007/3-540-47910-4_21`.

19 Google. Google Cloud Pub/Sub documentation [online]. URL: `https://cloud.google.com/pubsub/docs` [cited 2022-05-12].

20 Google. pytype [online]. URL: `https://google.github.io/pytype/` [cited 2021-11-30].

21 David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001. `doi:10.1145/506315.506316`.

22 Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 12–19. Springer, 2018. `doi:10.1007/978-3-319-96142-2_2`.

23 Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 152–162. ACM, 2018. `doi:10.1145/3236024.3236051`.

24 Maximilian A. Köhl. An executable structural operational formal semantics for Python. Master's thesis, Saarland University, December 2020. URL: `https://arxiv.org/abs/2109.03139`.

25 Jukka Lehtosalo, Guido van Rossum, Ivan Levkivskyi, and Michael J. Sullivan. mypy - optional static typing for Python [online]. URL: `http://mypy-lang.org/` [cited 2021-11-30].

26 Microsoft. Pyright: Static type checker for Python [online]. URL: `https://github.com/microsoft/pyright` [cited 2021-11-30].

27 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static type analysis by abstract interpretation of Python programs. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.ECOOP.2020.17`.

28 Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. Static analysis for AWS best practices in Python code. *CoRR*, abs/2205.04432, 2022. `doi:10.48550/arXiv.2205.04432`.

29 Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: the full monty. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 217–232. ACM, 2013. `doi:10.1145/2509136.2509536`.

30 Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: neural type prediction with search-based validation. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 209–220. ACM, 2020. doi:10.1145/3368089.3409715.

31 Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 111–124. ACM, 2015. doi:10.1145/2676726.2677009.

32 Michael Salib. *Starkiller : a static type inferencer and compiler for Python*. PhD thesis, Massachusetts Institute of Technology, May 2004.

33 Gideon Joachim Smeding. An executable operational semantics for Python. Master's thesis, Universiteit Utrecht, 2008. URL: http://www.cs.uu.nl/education/scripties/scriptie.php?SID=INF/SCR-2008-029.

34 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In Andrew P. Black and Laurence Tratt, editors, *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 45–56. ACM, 2014. doi:10.1145/2661088.2661101.

35 Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=Hkx6hANtwH.

36 Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 607–618. ACM, 2016. doi:10.1145/2950290.2950343.

What If We Don't Pop the Stack? The Return of 2nd-Class Values

Anxhelo Xhebraj
Purdue University, West Lafayette, IN, USA

Oliver Bračevac
Purdue University, West Lafayette, IN, USA

Guannan Wei
Purdue University, West Lafayette, IN, USA

Tiark Rompf
Purdue University, West Lafayette, IN, USA

Abstract

Using a stack for managing the local state of procedures as popularized by Algol is a simple but effective way to achieve a primitive form of automatic memory management. Hence, the call stack remains the backbone of most programming language runtimes to the present day. However, the appealing simplicity of the call stack model comes at the price of strictly enforced limitations: since every function return pops the stack, it is difficult to return stack-allocated data from a callee upwards to its caller – especially *variable-size* data such as closures.

This paper proposes a solution by introducing a small tweak to the usual stack semantics. We design a type system that tracks the underlying *storage mode* of values, and when a function returns a stack-allocated value, we *just don't pop the stack*! Instead, the stack frame is de-allocated together with a parent the next time a heap-allocated value or primitive is returned. We identify a range of use cases where this delayed-popping strategy is beneficial, ranging from closures to trait objects to other types of variable-size data. Our evaluation shows that this execution model reduces heap and GC pressure and recovers spatial locality of programs improving execution time between 10% and 25% with respect to standard execution.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Call stack, closures, stack allocation, memory management, 2nd-class values, capabilities, effects

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.15

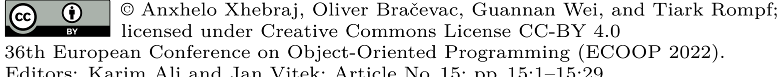
Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.26

Funding This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, and DOE award DE-SC0018050.

Acknowledgements We thank the anonymous reviewers for their insightful comments.

1 Introduction

Using a call stack to manage activation records of procedures was one of the great advances in the design and implementation of programming languages. Discovered by Bauer in the 1950s [11] and popularized by Dijkstra and others in the design of Algol in the 1960s [23, 29], the call stack enabled general recursion by supporting multiple concurrent activations of the same function [23], a significant gain in expressiveness over early Fortran dialects and other languages of the time. The call stack also provides a simple but effective form of automatic memory management, which makes it the backbone of almost every programming language runtime to the present day (with some notable exceptions, e.g., SML/NJ [4, 37]).



© Anxhelo Xhebraj, Oliver Bračevac, Guannan Wei, and Tiark Rompf;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 15; pp. 15:1–15:29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, the appealing simplicity of the call stack model comes at the price of strictly enforced limitations. Since *every* function return pops the stack, it is difficult to return stack-allocated data from a callee upwards to its caller – especially variable-size data such as closures. Existing language implementations sidestep this issue by allocating return values beyond a small fixed size on the heap (forgoing benefits of stack allocation such as guaranteed timely deallocation), by trying to infer or bound the size of returned values and reserving sufficient space in the parent stack frame (not always possible and sometimes wasteful), or by abandoning the rigid call stack altogether in favor of more flexible structures such as stacks of resizable regions (with overall increased complexity and loss of performance predictability).

This work is based on the observation that in cases where returning a stack-allocated value is desired, the value's lifetime is typically still bounded, it just needs to live *a little bit* longer than the procedure that allocated it. So, what would happen if we *just don't pop the stack* and delay it until one of the callers resets the stack, popping multiple frames at once? It turns out that this surprisingly simple idea can be made to work rather well. When returning a stack-allocated value, we retain the callee stack frame, essentially treating it as a block allocated as part of the parent stack frame. The stack is reset once a primitive or heap-allocated value is returned, resulting in joint deallocation of multiple stack frames. We present a type system that tracks the underlying storage type of references to guide reclamation decisions. The key safety invariant is that heap-allocated data does not contain stack references, and that stack references only point upward, not downward.

The central appeal of this "delayed popping" strategy is its simplicity and attractive power-to-weight ratio. Since the approach constitutes only a minor adjustment to the ubiquitous call stack model, it should be easy to add to almost any language implementation supporting stack allocation. All the necessary safety checks can be retrofitted onto existing type systems that track scoped lifetimes of values. In particular, we extend Osvald et al.'s [45] $\lambda^{1/2}$ type system for tracking 2nd-class values, achieving considerable gains in expressiveness (e.g., curried 2nd-class functions [16]). Delayed popping and returning variable-size data on the stack benefit a variety of uses cases, ranging from closures to trait objects to other types of data. Reducing heap and GC pressure and increasing spatial locality significantly improves end-to-end execution time. Certain classes of programs previously dominated by GC overhead can now run entirely on the stack.

In summary, this paper makes the following contributions:

- We discuss the problem of returning stack-allocated values and describe our solution of delaying stack frame reclamation informally using examples (Section 2).
- We present the $\lambda^{1/2}_{\hookleftarrow}$ type system for tracking *storage-mode* qualifiers. To ensure validity of references in the presence of both heap and stack references, we attach simple qualifiers to types [25] and restrict stack-allocated values to a *2nd-class* status (Section 3).
- We present the operational semantics of our model, and we establish key properties: (1) type safety: well-typed programs do not go wrong, (2) memory separation: 1st-class values do not refer to 2nd-class values, (3) stack allocation: 2nd-class values can be stack-allocated with delayed popping (Section 4). We have mechanized the proofs in Coq.
- We present compiler implementations of our approach (1) in Scala Native [50] with an LLVM backend using a shadow stack, and (2) in MiniScala with an x86-64 backend using the native call stack (Section 5).
- We provide in-depth discussions of extensions and uses cases, e.g., support for parametric polymorphism (including abstracting over storage modes), on-stack mutable data, programming with capabilities, and how we overcome limitations of previous work on 2nd-class values and existing language implementations (e.g., Rust) when returning variable-size data on the stack (Section 6).

- We evaluate our system on a variety of benchmarks showing speedups between 10% and 25% and present two in-depth case studies on static memory management for deep learning workloads and parser combinators (Section 7).

We discuss related work in Section 8 and conclude in Section 9. Our artifacts (implementation, Coq proofs) are available online at `https://github.com/angelogeb/scala-native`.

2 The Return of Stack-Allocated Values

Since the days of Algol [11, 23, 29], stack-allocated activation records (*stack frames*) are the core data structure for implementing multiple activations of the same function. Stack frames are instantiated when performing a function call and used for: (1) storing the return address of the calling procedure, (2) passing the parameters to the called procedure, (3) storing temporary local variables that are de-allocated at the end of the procedure, and (4) passing returned values back to the caller. In this paper we are interested in (3), the stack allocation of values, and (4), passing values back to the calling procedure on the stack.

The goal: returning stack-allocated data. While dealing with stack-allocated primitives of a fixed size (such as 4 or 8 bytes) is trivial, returning stack-allocated compound data types raises some issues. Consider computing the norm of the sum of two vectors $\|v_1 + v_2\|_2$ using a library that provides the functions `add` and `norm`:

```
norm(add(v1, v2))
```

The function `add` needs to return an array denoting the sum of `v1` and `v2`. However, returning an array typically requires allocating its storage on the heap, unless one is willing to copy a potentially sizable chunk of memory repeatedly between stack positions. Heap allocation is not ideal in this case either because the lifetime of the value is known: it is immediately consumed by `norm`. Moreover, the allocation will have to be reclaimed in ways that are unsafe (manually), have a performance penalty (garbage collection or reference counting) or require more complicated type systems based on lifetimes and borrowing.

Destination-passing style: a manual workaround. To work around this issue, code bases written in Fortran or C from the domain of High Performance Computing (HPC) often design functions such as `add` in *Destination-Passing Style* (DPS) [33, 51]. In this setting, every function that needs to return an array accepts a pre-allocated result buffer as argument, so that the caller is in control of the result's memory management:

```
@stack val vout = new Vec[f32](v1.length)
add(v1, v2, vout)
norm(vout)
```

We use Scala Native in this paper, but the discussion equally applies to any language that supports stack allocation. We use the `@stack val` ... annotation to denote stack-allocated values, building on a recently proposed type system by Osvald et al. [45] to guarantee that such "2nd-class" values do not escape. In C, the example could be equivalently written using a local array declaration or explicitly using the stack allocation primitive `alloca(v1.length * sizeof(float))`, although without any safety checking. We describe our model, including the type system, in more detail in Section 2.2. The DPS implementation of `norm` and `add` is shown in Figure 1a.

In the example above, the caller decides to allocate the result buffer on the stack, which means that the buffer will be de-allocated automatically at the end of the scope. However, this comes with downsides too – we lose the ability to write expression-oriented code like `norm(add(v1, v2))` or perhaps `norm(v1 + v2)` with proper operator overloading, and potential

sharing of buffers can lead to subtle bugs. For example, a call like `movingAverage(vin=v, vout=v)` would produce an incorrect value, since the output at index `i` depends on the input at index `i-1`, which would have been overwritten by the previous iteration:

```
def movingAverage(vin: Vec[f32] @stack, vout: Vec[f32] @stack) = {
  for (i <- 1 until (vin.length - 1))
    vout(i) = (vin(i - 1) + vin(i) + vin(i + 1)) / 3
}
```

Variable-size data: the key limitation of destination passing. In the case of `add`, it is possible for the caller to provision memory for the callee and allocate the result buffer on the stack, but there are cases where this is not possible and we have to rely on the heap instead. Consider a program that has to deserialize data using a utility function `readNextGroup` which reads a data-dependent number of bytes from a stream:

```
def readNextGroup(f: FStream) = {
  val len = readInt(f)    // data-dependent
  val res = new Vec[Byte](len) ① // alloc
  readBytes(res, len)
  res
}
```

```
def deserialize(f: FStream): Tree = {
  val buf = readNextGroup(f) // unknown result size:
  buf(0) match {             // can't pre-allocate!
    case I32_TAG => atoi(buf)
    ...
} }
```

Unlike the `add` example, the result's size is data dependent and therefore it is not possible to manage memory in DPS, unless we were to break modularity and function boundaries by inlining. Instead, the code must allocate the result buffer on the heap (①). Both examples need to work around the inability to *return short-lived dynamically sized* objects on the stack.

Closures: a particular important case of variable-size data. The sizing restriction of the strict stack discipline also conflicts with higher-order functions. Programming languages based on stack environments rely on types to compute a value's storage size. However, a closure's type does not describe its closing environment and, unlike the array examples, a solution attempt would neccessarily involve some form of defunctionalization [22] and intensional type analysis, again breaking modularity.

```
def useCurried(f: Int => Int => Int) = {
  val g = f(10) // unknown result size:
  ...           // can't pre-allocate!
}
```

```
val (x, y) = ...
useCurried(_ => (_ => 1)①)
useCurried(a => (b => a * x + b * y)②)
```

In the program above, the size of the closure `g` returned by `f` is statically unknown. In the first call, the returned closure (①) has a trivial size. In the second call, the returned closure (②) captures `x` and `y` and therefore has a larger size than (①). Pre-allocating memory for `g` would require knowing all possible closures that can be returned and reserving space required for the largest one. In most cases this is infeasible, therefore returning closures requires forgoing stack allocations and relying extensively on the heap (see also Section 6.7).

Problem summary: always popping the stack is inflexible. The root of the problem is that every function is required to pop its stack frame immediately when returning. Under a *strict stack discipline* it is possible to return a value on the stack *only if its size can be statically upper-bounded at the call site*. In such cases, storage for the returned value is reserved on the caller's stack frame, a reference to the storage is passed to the callee and a copy of the returned value is performed by the callee (DPS). This is necessary to ensure that a function return resets the top of the stack to what it was before the function call.

2.1 A Partial Solution: 2nd-class Values and Selective CPS Conversion

Before looking at ways to relax the strict invariant that every function must pop its stack frame immediately when returning, we consider strategies that inspired our solution.

Selective CPS conversion to extend stack lifetime. In the presence of higher-order functions, we can eschew the problem of returning by transforming programs to Continuation-Passing Style (CPS) [47]. Rather than passing a destination address to the caller as in DPS, we pass a callback (the continuation) to receive the result address. The CPS transformation enables rewriting programs that return function values into equivalent programs where functions appear only in argument positions or as operands in call expressions (in the style of Algol) [24], including all cases that return *compound data types of unknown size* [38].

We can rewrite function `add` in CPS as shown in Figure 1b. Operationally the CPS transformation delays the reclamation of `add`'s activation record to the point where the continuation `k` returns. Instead of returning a value on the stack, `add` accepts a continuation that can use the "returned" value in non-escaping fashion. Figure 1c illustrates the stack behavior after the CPS rewrite. Before the call to `add`, `run` pushes the arguments on the stack (①). In ②, `add` has allocated the array on the stack and passes its reference to `norm`. After `norm` returns, both `add`'s and `run`'s stack frame will be reclaimed (③).

This solution achieves our initial goal, i.e., `add` can now "return" variable-size data. Importantly, the CPS transformation should be selective [40, 49, 7] since transforming all definitions into CPS would eliminate the call stack altogether [8, 38]. However, the solution is not ideal because it requires (1) selective non-local transformations of the code, and (2) it might result in even more heap allocations if continuations are not dealt with properly. Nevertheless, the CPS version of the program shows what is required to return variable-size stack-allocated values – deferring the de-allocation of the stack frame of the callee and a type system that ensures validity of stack references.

Tracking non-escaping values in types. The $\lambda^{1/2}$ type system by Osvald et al. [45] allows to implement this pattern safely. Their system qualifies values as "2nd class" through the `@local` annotation on types (we use `@stack` to denote the same in this paper). Such 2nd-class values are not allowed to escape their defining scope, i.e., their lifetimes follow a strict stack discipline. The $\lambda^{1/2}$ type system enforces that:

1. 1st-class functions may not refer to 2nd-class values through free variables.
2. Functions may not return 2nd-class values.
3. 2nd-class values may not be stored in mutable variables or object fields.

This ensures that 2nd-class references are used in a non-escaping fashion. The key safety invariant is that heap-allocated data is "1st class" and does not contain stack references, and that stack references only point upward, not downward. CPS preserves the precise stack behavior of 2nd-class values [19]. Crucially, $\lambda^{1/2}$ *disallows* returning 2nd-class values to maintain strict stack safety. A key contribution of this paper is to lift this restriction in a sound way, leading to a model of delayed reclamation of stack frames that exhibits the same desired behavior as the selective CPS conversion, but in *direct style*.

2.2 Our Solution: Delay Popping in Direct Style, Using Type Qualifiers

Based on the preceding insights, we propose a model to allow returning short-lived stack-allocated values for which sizes are unknown at compile time for programs *in direct style*, by allowing functions to return references to their stack frame and using Osvald-style [45] storage-mode qualifiers to guide stack reclamation.

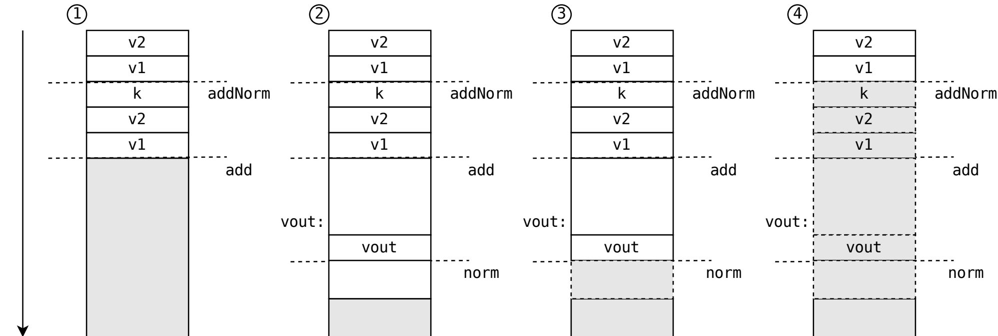
Figure 1d shows the implementation of `addNorm` in direct style using storage modes. When `add` returns `vout`, its stack frame is retained, relaxing the *too strict* stack discipline. To this end, we enrich Osvald et al.'s type system [45] with `@stack` qualifiers *on function return types*. These qualifiers drive the operational behavior in crucial ways:

```
def norm(v: Vec[f32] @stack): f32 = ...
def add(v1 : Vec[f32] @stack, v2: Vec[f32] @stack, res: Vec[f32] @stack): Unit = {
  for (i <- 0 until v1.length)
    res(i) = v1(i) + v2(i)
}
def addNorm(v1: Vec[f32] @stack, v2: Vec[f32] @stack): f32 = {
  @stack val vout = new Vec[f32](v1.length)
  add(v1, v2, vout)
  norm(vout)
}
```

(a) Destination passing style: having the caller preallocate storage is a standard workaround for "returning" stack-allocated values. This approach only works if the caller knows the (maximum) size of the result.

```
def add[T](v1: Vec[f32] @stack, v2: Vec[f32] @stack, k: (Vec[f32] @stack => T) @stack): T = {①
  @stack val vout = new Vec[f32](v1.length)
  for (i <- v1.length)
    vout(i) = v1(i) + v2(i)
  k(vout) ②
}
def addNorm(v1: Vec[f32] @stack, v2: Vec[f32] @stack): f32 = {
  add(v1, v2, norm ③)
} ④
```

(b) "Returning" a value on the stack through a selective CPS transform. The stack is managed as depicted in (c). Before the call to `add`, `addNorm` pushes the arguments on the stack (①). In ② `add` allocates the array on the stack and passes its reference to `norm`. After `norm`'s return (③), both `add`'s and `addNorm`'s stack frame will be reclaimed (④).



(c) Stack behavior of selective CPS version (b) when `addNorm` is called with references to the argument arrays `v1`, `v2` on the stack: right after entering `add`'s body (①), after calling the continuation `norm` (②), after `norm`'s return (③) and after `addNorm`'s return (④). Grey portions of the stack are free.

```
def add(v1: Vec[f32] @stack, v2: Vec[f32] @stack): Vec[f32] @stack = { ①
  @stack val vout = new Vec[f32](v1.length)
  for (i <- v1.length)
    vout(i) = v1(i) + v2(i)
  vout
}
def addNorm(v1: Vec[f32] @stack, v2: Vec[f32] @stack): f32 = {
  norm(add(v1, v2)②)③
} ④
```

(d) Returning a value on the stack through storage modes. After ①, `add`'s stack frame is retained to later be popped when `addNorm` returns (④). The stack behavior is the one depicted in (c) without `k`.

.

Figure 1 Alternatives for stack-allocated arrays: DPS (a), selective CPS (b) and its stack behavior (c), direct style with storage modes (d). Storage modes emulate the selective CPS stack behavior.

1. When returning a 2nd-class value, do not pop the stack.
2. In that situation, defer the deallocation of the callee's stack frame to a point where a caller further up the stack returns a 1st class value.
3. When returning a 1st-class value, reset the stack as usual; this will reclaim all stack frames allocated by callees.

This strategy is safe because 1st-class values cannot refer to 2nd-class ones, and 2nd-class values do not escape other than through the return path, emulating the CPS stack behavior (Figure 1). We break with the convention that the top of the stack remains unchanged after function applications returning `@stack`-qualified values. It is important to note that this does not interfere in major ways with the caller's stack layout. As long as the caller maintains a pointer to the start of its own stack frame, it can treat the remaining callee stack frame like any other piece of stack-allocated data and continue allocating at the current stack pointer, pointing to the end of the stack frame (see also Section 5).

Controlling allocations through storage-mode qualifiers. We provide further examples to showcase our programming model. We assume a call-by-value language with primitive types, compound types (e.g., closures and arrays), and automatic memory management, representative of languages like Java, OCaml, or Scala. Values are either constants of primitive types or references to values of compound types (either allocated on the stack or a heap). Closures capture the smallest environment by value and store it in the closure object.

Stack bindings and storage modes. Values can be allocated on the stack through `@stack` bindings. In the following example `inc1` is a stack-allocated closure:

```
def add1(l: List[Int]) = {      // : List[Int] => List[Int]
  @stack val inc1 = i => i + 1  // : (Int => Int) @stack
  map(l, inc1)                  // : List[Int]
}
add1(List(1,2,3,4))             // = List(2,3,4,5)
```

Stack bindings induce a `@stack` annotation *on the type* of the bound variable, called *storage-mode qualifiers*. Stack bindings can be omitted if they can be inferred from types, e.g., we can equivalently annotate `inc1` with a type ascription:

```
val inc1: (Int => Int) @stack = i => i + 1
```

where the `@stack` qualifier is attached to the function type. We can also rewrite the body of `add1` more concisely as

```
map(l, i => i + 1)              // stack allocation inferred
```

and let type inference assign the desired storage mode to the function argument of `map`, as explained next.

Storage-mode statics. While unannotated values (1st-class values) can be used freely, the type system ensures that `@stack`-qualified values do not escape into 1st class contexts, i.e., non-`@stack`-qualified positions. For the example above, the type of `map`'s second argument *must* be `@stack` qualified as shown below (①):

```
def map[I, O](l: List[I], f: (I => O) @stack①): List[O] = l match {
  case Nil => Nil
  case Cons(h, t) => Cons(f(h), map(t, f))
}
```

However, the `@stack` qualifier only restricts the uses of `f` but does not mandatorily induce a stack allocation, i.e., we can still pass heap-allocated functions to `map` (see also Section 6.2):

```
val f = (v: Int) => v + 10      // closure allocated on the heap
map(l, f)                       // ok
```

```
def filter(l: List[Int] @stack, f: (Int => Boolean) @stack): List[Int] @stack = l match {
  case Nil => Nil
  case Cons(h, t) =>
    val t' = filter(t, f)
    if (f(h)) Cons(h, t') else t'
}
def run(l: List[Int]): Boolean = { ①
   val l' = filter(l, x % 2 == 0)
② sum(l') > 0
}
val l = List(1,2,3,4,5)
run(l)
```

Figure 2 Example program that returns a list on the stack. `filter`'s stack frame is retained after its return (shown in cyan). Note that `filter` also works with heap-allocated list arguments.

This is achieved by the subtyping relationship `@heap` `<:` `@stack` where unqualified types `T` have an implicit qualifier `@heap` (1st class). Storage modes and their static semantics ensure that *heap-allocated objects do not refer to stack-allocated ones.*

Storage-mode dynamics. Storage modes permit returning variable-size data on the stack by retaining the callee's stack frame when it returns a `@stack`-qualified value. The stack frame is popped once reaching a caller returning a `@heap`-qualified value. We support returning any type of variable-size data, even recursive ones such as lists (Figure 2), the `filter` function builds the result list on the stack. After the call to `filter`, the execution returns to `run` with the updated stack pointer (②). The returned list is passed as an argument to `sum`, which will execute its body and pop its stack frame. Once `run` returns, it will pop its stack frame, too (since it returns a `@heap`-qualified value[1]), which also contains `filter`'s stack frame.

Values with a `@stack` qualifier are 2nd class. As noted above, storage modes are an instance of Osvald et al.'s 2nd-class values [45]. Their work proposed to reintroduce 2nd-class values in the style of Algol's procedure parameters to implement capabilities for safe exceptions and, more generally, effect checking. To achieve this, their type system accepts only function definitions that have 1st-class return types to ensure that capabilities conform to the strict stack discipline. However, this is restrictive and inhibits many use cases where a returned value has to close over a 2nd-class argument, such as currying over a capability.

Our work removes this limitation and employs qualifiers, not only to ensure the validity of references but also to change the runtime behavior of the stack.

3 The $\lambda^{1/2}_{\hookleftarrow}$ Storage-Mode Qualifier Calculus

We formalize storage modes in the $\lambda^{1/2}_{\hookleftarrow}$-calculus, an extension of the $\lambda^{1/2}$-calculus by Osvald et al. [45], which distinguishes between 1st-class and 2nd-class values.[2] In the $\lambda^{1/2}$ system, functions cannot return 2nd-class values, implying strict stack-based 2nd-class lifetimes. Our $\lambda^{1/2}_{\hookleftarrow}$-calculus eliminates this restriction, yielding considerable gains in expressiveness, e.g, it supports currying of functions that take 2nd-class arguments, whereas $\lambda^{1/2}$ does not. We prove type soundness for $\lambda^{1/2}_{\hookleftarrow}$ with respect to the standard call-by-value semantics of the λ-calculus. This lays the groundwork for proving stronger results about memory invariants, such as safety and correctness of stack allocation with delayed popping (Section 4).[3]

[1] Scalar primitives are always 1st class (`@heap` qualified) and therefore can be used freely.
[2] Our system also scales to $F_{<:}$-style parametric polymorphism using path-dependent types (cf. [61]).
[3] We mechanized these result in Coq, available at `https://github.com/angelogeb/scala-native`.

Syntax $\boxed{\lambda^{1/2}_{\hookleftarrow}}$

$q ::= 1 \mid 2 \qquad t ::= c \mid x \mid \lambda x.t \mid t\ t \qquad T ::= B \mid T^q \to T^{q} \qquad \Gamma ::= \varnothing \mid \Gamma, x : T^q$

$$\Gamma^{[\leq q]} := \{x : T^{q_x} \in \Gamma \mid q_x \leq q\}$$

Subtyping Rules $\boxed{T^{q_1} <: T^{q_2}}$

$$\frac{q_1 \leq q_2}{B^{q_1} <: B^{q_2}}\ \textsc{SRfl} \qquad \frac{T_3^{q_3} <: T_1^{q_1} \quad T_2^{q_2} <: T_4^{q_4} \quad q_5 \leq q_6}{(T_1^{q_1} \to T_2^{q_2})^{q_5} <: (T_3^{q_3} \to T_4^{q_4})^{q_6}}\ \textsc{SFun}$$

Typing Rules $\boxed{\Gamma \vdash t : T^q}$

$$\frac{}{\Gamma \vdash c : B^q}\ \textsc{TCst} \qquad \frac{\Gamma(x) = T^q}{\Gamma \vdash x : T^q}\ \textsc{TVar} \qquad \frac{\Gamma^{[\leq q]}, x : T_1^{q_1} \vdash t : T_2^{q_2}}{\Gamma \vdash \lambda x.t : (T_1^{q_1} \to T_2^{q_2})^q}\ \textsc{TAbs}$$

$$\frac{\Gamma \vdash t_1 : (T_1^{q_1} \to T_2^{q_2})^2 \quad \Gamma \vdash t_2 : T_1^{q_1}}{\Gamma \vdash t_1\, t_2 : T_2^{q_2}}\ \textsc{TApp} \qquad \frac{\Gamma \vdash t : T_1^{q_1} \quad T_1^{q_1} <: T_2^{q_2}}{\Gamma \vdash t : T_2^{q_2}}\ \textsc{TSub}$$

Figure 3 The $\lambda^{1/2}_{\hookleftarrow}$ type system. Differences to Osvald et al.'s system [45] are highlighted.

3.1 Syntax and Typing Rules

Terms in $\lambda^{1/2}_{\hookleftarrow}$ (Figure 3) can be constant literals of base types, variables, functions, and applications. We annotate types with storage-mode qualifiers q. These can be "1" denoting a 1st-class/heap-allocated result, or "2" denoting a 2nd-class/stack-allocated result. Storage modes are totally ordered, where $1 \leq 2$.

Typing environments Γ (Figure 3) come with the usual *lookup* operator $\Gamma(x) = T^q$ and a *filter* operator $\Gamma^{[\leq q]}$ that returns all the assumptions $x : T^{q_x}$ in Γ satisfying $q_x \leq q$.

The type system (Figure 3) is an extension of the simply-typed λ-calculus (STLC) with subtyping plus storage-mode qualifiers on types. Qualifiers propagate through the subtyping rules, essentially allowing the use of 1st-class terms in positions where 2nd-class terms are expected (SRfl), and they are subject to the usual contravariance in function domains and covariance in function codomains (SFun). Filtering the typing context when typing function bodies (TAbs) ensures that 1st-class functions do not capture 2nd-class values.[4]

The subtle yet crucial difference to Osvald et al.'s $\lambda^{1/2}$-calculus is that $\lambda^{1/2}_{\hookleftarrow}$ also attaches qualifiers to *codomains* of function types, granting more degrees of freedom when typing functions (TAbs) and applications (TApp); e.g., this enables currying of 2nd-class parameters.

3.2 Type Soundness of Standard Small-Step Evaluation

We use the standard call-by-value (cbv) single-step reduction of the λ-calculus, and prove type soundness via progress and preservation lemmas [60]. The proofs require reflexivity and transitivity of subtyping, as well as weakening and narrowing lemmas, which are standard.

▶ **Theorem 1** (Progress). *Given a closed term t that is well-typed $\varnothing \vdash t : T^q$ then either t is a value or else there exists t' such that $t \longrightarrow t'$.*

Proof. By induction on the derivation $\varnothing \vdash t : T^q$, using canonical forms lemmas. ◀

[4] For simplicity, the base system lacks recursive functions. These can be readily added by allowing self-references in lambdas ($\lambda f(x).t$), adding $f : (T_1^{q_1} \to T_2^{q_2})^q$ to the body's context in TAbs, and generalizing TApp to make use of q.

Big-Step Evaluation: Standard and Instrumented $\boxed{\mathcal{H} \vdash t \Downarrow^q v}$

$$t ::= c \mid x^q \mid \lambda x^q.t^q \mid t\ t \qquad v ::= c \mid \langle \mathcal{H}, \lambda x^{q_1}.t^{q_2} \rangle \qquad \mathcal{H} ::= \varnothing \mid \mathcal{H}, x^q : v$$

$$\mathcal{H}^{[\leq q]} := \{(x^{q_x} : v) \in \mathcal{H} \mid q_x \leq q\}$$

$$\frac{}{\mathcal{H} \vdash c \Downarrow^q c}\ \text{QECst} \qquad \frac{\mathcal{H}^{[\leq q]}(x^{q_1}) = v}{\mathcal{H} \vdash x^{q_1} \Downarrow^q v}\ \text{QEVar} \qquad \frac{}{\mathcal{H} \vdash \lambda x^{q_1}.t^{q_2} \Downarrow^q \langle \mathcal{H}^{[\leq q]}, \lambda x^{q_1}.t^{q_2} \rangle}\ \text{QEAbs}$$

$$\frac{\mathcal{H} \vdash t_1 \Downarrow^2 \langle \mathcal{H}', \lambda x^{q_2}.t_3^{q_3} \rangle \qquad \mathcal{H} \vdash t_2 \Downarrow^{q_2} v_2 \qquad \mathcal{H}', x^{q_2} : v_2 \vdash t_3 \Downarrow^{q_3} v_3 \qquad q_3 \leq q}{\mathcal{H} \vdash t_1\, t_2 \Downarrow^q v_3}\ \text{QEApp}$$

Figure 4 Two big-step semantics for $\lambda^{1/2}_{\hookleftarrow}$ using environments. (1) Excluding **teal** parts: the standard call-by-value big-step semantics of the λ-calculus. (2) Including **teal** parts: a more restrictive semantics that internalizes storage modes in the term syntax and checks storage modes.

▶ **Theorem 2** (Preservation). *Given a closed term t that is well-typed with type T^q, i.e., $\varnothing \vdash t : T^q$, if t steps to t', $t \longrightarrow t'$, then $\varnothing \vdash t' : T^q$.*

Proof. By induction on the derivation $\varnothing \vdash t : T^q$, using the usual lemmas for inversion of typing and preservation under substitution. ◀

From progress and preservation, we can establish type soundness of the evaluation semantics induced by the single-step reduction relation [60].

4 Memory Properties

We have established in Section 3.2 that "well-typed $\lambda^{1/2}_{\hookleftarrow}$ programs do not go wrong" in terms of the cbv λ-calculus reduction semantics. While already important, this only asserts that well-typed $\lambda^{1/2}_{\hookleftarrow}$ terms do not exhibit the runtime errors of the ordinary λ-calculus. However, we need to prove further memory properties: (1) 1st-class values never close over 2nd-class values, and (2) delayed popping of the stack is safe. The solution is *refining* the semantics to check for *more* runtime errors, and prove that type soundness still holds.

4.1 1st-Class Values Never Capture 2nd-Class Values

As a first refinement, we let the term syntax carry explicit qualifier annotations q (obtainable from typing derivations) and define a new "instrumented" evaluation semantics checking for class violations at runtime, resulting in the calculus $\lambda^{1/2}_{q\hookleftarrow}$ (Figure 4, semantics 2). It augments the standard big-step semantics of the cbv λ-calculus with the colored parts.

We check for qualifier mismatches in lookup (rule QEVar) and that the qualifier of a closure's function body agrees with the one under evaluation (rule QEApp). When building closures in rule QEAbs, only a subset of the environment is captured, which is enforced by filtering the environment with the current class context q. These changes (1) implicitly partition $\mathcal{H}$ into a 1st- and 2nd-class environment, and (2) make evaluation stuck if a 1st-class/heap-allocated value captures a 2nd-class/stack-allocated value.

Proof technique. We model Wright and Felleisen's "strong soundness" notion [60] using a total evaluator function[5] in the style of Siek [52] and Amin and Rompf [3]

$$\mathsf{eval}^q : \mathbb{N} \to \mathcal{H} \to t \to (\mathsf{Done}\ (\mathsf{Val}\ v \mid \mathsf{Wrong}) \mid \mathsf{Timeout})$$

which, given a fuel value $k \in \mathbb{N}$ and a runtime environment $\mathcal{H}$, evaluates a term to a result r that can be either (1) Timeout if the fuel is not enough to complete the evaluation, or (2) Done Wrong in case of a runtime error, or (3) Done (Val v) for a result value v. Here, eval^q implements the big-step evaluation relation $\mathcal{H} \vdash t \Downarrow^q v$ (Figure 4), i.e., $\mathcal{H} \vdash t \Downarrow^q v$ if and only if there is a fuel value $k \in \mathbb{N}$ such that $\mathsf{eval}^q\ k\ \mathcal{H}\ t = \mathsf{Done}\ (\mathsf{Val}\ v)$.

Type soundness implies memory separation. The strong soundness proof depends on well-typed values ($v : T^q$) and well-formed environments ($\Gamma \vDash \mathcal{H}$), defined below. The key property is demanding that well-typed closures capture *only* values below or at their assigned class, *and nothing else*, as follows:

$$\frac{}{c : B^q} \qquad \frac{\Gamma \vDash \mathcal{H} \quad \mathcal{H}^{[\leq q]} = \mathcal{H} \quad \Gamma, x : T_1^{q_1} \vdash t : T_2^{q_2}}{\langle \mathcal{H}, \lambda x^{q_1}.t^{q_2} \rangle : (T_1^{q_1} \to T_2^{q_2})^q} \qquad \frac{}{\varnothing \vDash \varnothing} \qquad \frac{\Gamma \vDash \mathcal{H} \quad v : T^q}{\Gamma, x : T^q \vDash \mathcal{H}, x^q : v}$$

▶ **Theorem 3** (Strong Soundness). *The $\lambda_{\leftrightarrows}^{1/2}$ type system is sound with respect to the instrumented big-step semantics (Figure 4, semantics 2): For all q, and for all k, if eval^q does not time out, then its result is a well-typed value:*

$$\frac{\Gamma \vdash t : T^q \qquad \Gamma \vDash \mathcal{H} \qquad \mathsf{eval}^q\ k\ \mathcal{H}\ t' = \mathsf{Done}\ r}{r = \mathsf{Val}\ v \qquad v : T^q}$$

Proof. By induction on the fuel value k, and case analysis on the term t, using helper lemmas to establish soundness of environment lookup. ◀

Intuitively, due to value typing and the extra class violation checks in the instrumented semantics, the strong soundness Theorem 3 implies:

▶ **Corollary 4.** *Well-typed 1st-class functions never capture 2nd-class values.*

Proof. By Theorem 3 and the definition of well-typed values for 1st-class function types. ◀

4.2 Stack-based Evaluation with Deferred Popping is Safe

As a further refinement, we design a semantics where 2nd-class bindings follow a delayed stack discipline and thus permit a corresponding practical call-stack implementation. Figure 5 shows the evaluation rules of the refined semantics with stacks. The big-step relation $\mathsf{H}, \mathsf{S} \vdash t \Downarrow_{\mathsf{s}}^q v \dashv \mathsf{S}'$ accepts as input an environment H, a stack S, a term t, and qualifier q, producing an output value v and a new stack S'. The stack is effectively a piece of state, threaded through computations. An environment H is an association list as usual while a stack S is a list of frames, where a frame Φ is also an association list of bindings. Stacks are *snoc* lists, with the head element having the largest index. Occasionally, we use the notation $\Phi_0 \ldots \Phi_k$ to visualize the stack and the corresponding index of each frame. The environment lookup $(\mathsf{H}, \Phi)(x^q)$ depends on the variable's qualifier q (rules EVarH and EVarS), e.g., we look up 2nd-class variables in the topmost stack frame. The same applies for environment extension $(\mathsf{H}, \Phi) \oplus x^q : v$ (cf. [61] for their formal definitions).

[5] This proof style is more succinct for proving the sought-after runtime invariants, because it models closures explicitly. The switch to big step is justifiable, because small- vs. big-step semantics, and substitution- vs. environment-based semantics are known to be equivalent [13, 1, 20].

Stack-based Big-Step Evaluation $\boxed{\mathsf{H},\mathsf{S} \vdash t \Downarrow^q_{\mathsf{s}} v \dashv \mathsf{S}}$

$$
\begin{array}{rrcl}
\text{Pointer} & ptr & ::= & k \mid \bot \\
\text{Frames/Env.} & \Phi, \mathsf{H} & ::= & \mathsf{L} \\
\text{Value} & v & ::= & c \mid \langle \mathsf{H}, ptr, \lambda x^{q_1}.t^{q_2} \rangle
\end{array}
\qquad
\begin{array}{rrcl}
\text{List} & \mathsf{L}_k & ::= & \varnothing_{-1} \mid (\mathsf{L}_{k-1}, x^q : v)_k \\
\text{Stack} & \mathsf{S} & ::= & \varnothing \mid (\mathsf{S}, \Phi) \\
 & k, i & \in & \mathbb{N}
\end{array}
$$

$$
\text{ECST}\quad \frac{}{\mathsf{H},\mathsf{S} \vdash c \Downarrow^q_{\mathsf{s}} c \dashv \mathsf{S}}
\qquad
\text{EVARH}\quad \frac{(\mathsf{H},\varnothing)(x^1) = v}{\mathsf{H},\mathsf{S} \vdash x^1 \Downarrow^1_{\mathsf{s}} v \dashv \mathsf{S}}
$$

$$
\text{EVARS}\quad \frac{(\mathsf{H},\Phi)(x^q) = v}{\mathsf{H},(\mathsf{S},\Phi) \vdash x^q \Downarrow^2_{\mathsf{s}} v \dashv (\mathsf{S},\Phi)}
$$

$$
\text{EABSH}\quad \frac{}{\mathsf{H},\mathsf{S} \vdash \lambda x^{q_1}.t^{q_2} \Downarrow^1_{\mathsf{s}} \langle \mathsf{H}, \bot, \lambda x^{q_1}.t^{q_2} \rangle \dashv \mathsf{S}}
$$

$$
\text{EABSS}\quad \frac{}{\mathsf{H},(\mathsf{S},\Phi) \vdash \lambda x^{q_1}.t^{q_2} \Downarrow^2_{\mathsf{s}} \langle \mathsf{H}, |\mathsf{S}|, \lambda x^{q_1}.t^{q_2} \rangle \dashv (\mathsf{S},\Phi)}
$$

$$
\text{EAPPH}\quad \frac{\begin{array}{c}\mathsf{H},\mathsf{S} \vdash t_1 \Downarrow^2_{\mathsf{s}} \langle \mathsf{H}', ptr, \lambda x^{q_2}.t_3^{1} \rangle \dashv \mathsf{S}' \\ \Phi = \mathit{lookup}(\mathsf{S}', ptr) \\ \mathsf{H},\mathsf{S}' \vdash t_2 \Downarrow^{q_2}_{\mathsf{s}} v_2 \dashv \mathsf{S}'' \\ (\mathsf{H},(\mathsf{S}'',\Phi)) \oplus x^{q_2} : v_2 \vdash t_3 \Downarrow^1_{\mathsf{s}} v_3 \dashv \mathsf{S}''' \end{array}}{\mathsf{H},\mathsf{S} \vdash t_1\, t_2 \Downarrow^1_{\mathsf{s}} v_3 \dashv \mathsf{S}}
$$

$$
\text{EAPPS}\quad \frac{\begin{array}{c}\mathsf{H},\mathsf{S} \vdash t_1 \Downarrow^2_{\mathsf{s}} \langle \mathsf{H}', ptr, \lambda x^{q_2}.t_3^{q_3} \rangle \dashv \mathsf{S}' \\ \Phi = \mathit{lookup}(\mathsf{S}', ptr) \\ \mathsf{H},\mathsf{S}' \vdash t_2 \Downarrow^{q_2}_{\mathsf{s}} v_2 \dashv \mathsf{S}'' \\ (\mathsf{H},(\mathsf{S}'',\Phi)) \oplus x^{q_2} : v_2 \vdash t_3 \Downarrow^{q_3}_{\mathsf{s}} v_3 \dashv \mathsf{S}''' \end{array}}{\mathsf{H},\mathsf{S} \vdash t_1\, t_2 \Downarrow^2_{\mathsf{s}} v_3 \dashv \mathsf{S}'''}
$$

Figure 5 $\lambda^{1/2}_{q\hookleftarrow}$ big-step stack semantics. Important details regarding evaluation are highlighted: (1) the evaluation relation is classified with a qualifier; (2) closures retain a pointer indicating the stack they capture; (3) stack is also an "output" of the relation and is not popped in rule EAPPS.

In contrast to the instrumented semantics (Figure 4), closures now contain a pointer *ptr* into the stack, which can be either $\bot$ (1st-class closures) or a natural number for the k-th stack frame (2nd-class closures). The $\mathit{lookup}(\mathsf{S}, ptr)$ operator (EAPPH, EAPPS) retrieves the k-th stack frame in S, if $ptr = k \in \mathbb{N}$, and otherwise a fresh stack frame if $ptr = \bot$.

Most rules are similar to their counterparts in Figure 4 and only read the input stack without updating it. When evaluating a 2nd-class function (EABSS), its closure records the pointer $|\mathsf{S}|$ to the topmost frame. In contrast, a 1st-class function is not supposed to close over 2nd-class values and thus does not retain a pointer ($\bot$ in EABSH). Both rules for 1st- and 2nd-class application (EAPPH and EAPPS) look up the closure's stack frame given its pointer and push it on top of the current stack to evaluate the function body. 1st-class application (EAPPH) pops the stack after evaluating the body, i.e., the output stack equals the input stack S. In contrast, 2nd-class application (EAPPH) *just does not pop the stack*, since the result v_3 might close over new 2nd-class bindings introduced during the body's evaluation. We define eval^q_s as the fuel-based interpreter corresponding to the stack-based evaluation relation.

Equivalence of stack and environment semantics implies safety. It is easy to recognize that the two semantics are equivalent in the sense that 2nd-class bindings are factored out of a common environment into an explicit stack and can only be captured through a pointer by closures. To this end, we define an equivalence relation $\mathsf{S} \vdash r_1 \sim r_2$ which relates the value, error, and divergence cases of the two semantics under a stack S. An environment $\mathcal{H}$ is equivalent to H, Φ if all of its values are equivalent to the values in the stack environment and vice versa (cf. [61] for the formal definition).

Equivalence is with respect to a stack because values in the stack semantics may capture 2nd-class references through stack pointers. Therefore, to relate them to values of the instrumented semantics, we must look up the right stack frame.

▶ **Theorem 5** (Equivalence of eval^q and eval^q_s)**.** *The instrumented environment semantics (Figure 4) and stack semantics (Figure 5) are equivalent, including timeout and error cases:*

$$\frac{(\mathsf{S},\Phi) \vdash \mathcal{H} \sim \mathsf{H},\Phi \qquad \mathsf{eval}^q\ k\ \mathcal{H}\ t\ = r_1 \qquad \mathsf{eval}^q_s\ k\ \mathsf{H}\ (\mathsf{S},\Phi)\ t\ = (r_2,\ \mathsf{S}')}{\mathsf{S}' \vdash r_1 \sim r_2}$$

Proof. By induction on fuel value k and case analysis on term t. ◀

▶ **Corollary 6** (Strong Soundness)**.** *The $\lambda^{1/2}_{\hookleftarrow}$ type system is sound with respect to the stack-based evaluation semantics with delayed popping (Figure 5).*

Proof. By the soundness Theorem 3 for the instrumented semantics and the equivalence Theorem 5, well-typed $\lambda^{1/2}_{\hookleftarrow}$ terms never evaluate to Wrong. ◀

Equivalence and soundness imply that well-typed $\lambda^{1/2}_{\hookleftarrow}$ terms exhibit all the desired memory properties in the stack-based semantics. It is thus safe to "just not pop the stack":

▶ **Corollary 7** (Separation of environment and stack)**.** *Evaluating well-typed $\lambda^{1/2}_{\hookleftarrow}$ terms never leaks stack references: If $\mathsf{eval}^q_s\ k\ \mathsf{H}\ (\mathsf{S},\Phi)\ t\ = (r,\ \mathsf{S}')$ for well-formed H and (S,Φ), then (1) all H' encountered during evaluation contain no stack pointers, and (2) if $r = \mathsf{Done}\ (\mathsf{Val}\ v)$, then all stack pointers in v are valid in S'.*

Hence, 2nd-class bindings can be safely implemented using a deferred stack discipline.

5 Implementation

We implement our system as an extension of Scala Native [50], a compiler backend for Scala that produces native code using LLVM [34]. Memory is managed at runtime by a non-copying variant of the Immix garbage collector [14]. We also implement our system in the MiniScala compiler used for teaching compiler classes at the authors' institution.

5.1 Scala Native

Scala Native compiles to LLVM which implements a fixed set of calling conventions and prohibits stack manipulation. Instead of allocating the `@stack` values on the system stack we rely on a *shadow stack*.

Shadow stack. Using a shadow stack simplifies the implementation, and allows us to implement the allocation scheme in any language and runtime. In addition, it optimizes memory usage of functions returning stack-qualified values, since only the returned value's storage is retained, while all other temporary stack values are reclaimed as usual.

The code generation is type directed but fairly simple: on entering functions that return a heap-qualified value we, instantiate a new stack frame in the shadow stack by first saving the previous top of the stack and then marking the new one. Shadow-stack allocations store the value in the current top of the stack, updating the stack pointer. When returning a heap-qualified value, the top of the shadow stack is reset to the top when entering the function. Since only frames for functions that return heap-qualified values are instantiated, calls to functions that return a stack-qualified value will update the stack pointer. In other words, our allocation strategy is similar to inlining regarding memory effects. Shadow-stack operations are inserted in a source Scala program as a source-to-source transformation. We insert calls for marking the top of the stack and resetting it.

Type system. We implement our type system as a compiler plugin for Scala's type system. The `@stack` qualifier is defined through the more general `@mode` annotation as shown below:

```
class mode[T] extends TypeConstraint
type stack = mode[Any];  type heap = mode[Nothing]
```

When type checking `T @mode[Q1] <: U @mode[Q2]` the type checker checks `T <: U` and `Q1 <: Q2` similarly to what is shown in the typing rules (Figure 3). Further implementation aspects (e.g., function signatures with qualifiers) closely follow those in [45]. The `@mode` annotation can be used to implement forms of storage-mode polymorphism (cf. Section 6.2).

5.2 MiniScala

MiniScala is a language and compiler which is used for teaching the compiler classes at the authors' institution. It differs from Scala Native in directly generating x86-64 assembly instead of LLVM, and in an overall *greatly* reduced feature set suitable for education, with a much simpler implementation of the type system and other components.

Native call stack. Following standard x86-64 conventions, the caller maintains a frame pointer (FP) pointing to the start of its own stack frame. The stack pointer (SP) points to the end of the stack, marking the point where fresh allocations can occur. Local variables are adressed FP-relative, i.e., as offsets of the frame pointer. Using this setup, a caller can treat a callee stack frame remaining after a call like any other piece of stack-allocated data, and continue allocating at the current stack pointer. Popping the caller stack frame will reset both FP and SP, and therefore reclaim all embedded callee stack frames, too.

Type system. In contrast to Scala Native, which can infer types based on bidirectional constraint propagation and resolution [44], MiniScala relies exclusively on a straightforward bidirectional typing algorithm without constraint generation. The simplicity of the algorithm means that generally more type annotations are required in user code than in full Scala.

Storage-mode polymorphism. MiniScala uses a simple erasure implementation of generics, and supports storage-mode polymorphism using allocator functions, but does not implement specific support to manage stack growth in generic code paths. Runtime dispatch on storage-mode witnesses could be added with reasonable implementation effort (cf. Section 6.2).

6 Discussion and Extensions

6.1 Tail calls

The stack-based semantics (Figure 5) does not model tail calls for simplicity and uniformity, i.e., function calls always create a fresh stack frame on top of the current one (EAppH and EAppS). However, it is possible to propagate tail contexts together with the qualifier q. Tail contexts calling a closure with a 1st-class argument can reuse the stack from its creation time, which is accessible from the captured stack pointer. Thus, we retain constant-space tail calls. This is safe since (1) all captured values at the closure's creation time are present, and (2) 1st-class parameters do not capture stack bindings. Tail calls with 2nd-class arguments cannot be optimized this way, as the argument may have been allocated anywhere on the stack, including in the current frame.

6.2 Storage-Mode Polymorphism

Basic subtyping. The subtyping relationship `@heap <: @stack` already provides *some* degree of polymorphism, allowing us to call functions accepting `@stack` parameters with `@heap` arguments. This encourages annotating non-escaping arguments as `@stack` since callers can pass both `@heap`- or `@stack`-qualified values. But naive subtype polymorphism has drawbacks: If we declare a function to return `@stack`, then (1) this does not guarantee that the return value is actually allocated on the stack, and (2) function returns cannot pop the stack anymore. It is thus desirable to introduce a proper notion of storage-mode polymorphism.

Parametric polymorphism. Having additional degrees of polymorphism is useful for two reasons: (1) dealing with higher-order functions, and (2) parameterizing where specific values are allocated. For example, we would like to generalize vector addition (Figure 1d) so that its result is allocated on the stack or on the heap. For case (1), we can use $F_{<:}$-style parametric polymorphism for storage modes, which is readily available in Scala's type system. We rephrase `@heap` and `@stack` as `@mode[Q]`, where `Q` ∈ `{Heap, Stack}`. For example, consider the higher-order function `logged`:

```
def logged[Q](f: (Int => Int) @mode[Q]): (Int => Int) @mode[Q] =
  { x => val res = f(x); println(res); res }
```

The type parameter `Q` abstracts over the storage mode in the function's signature, and we can refer to it using the `@mode[Q]` annotation. Reusing Scala's type language for storage-mode polymorphism permits expressing more complex relationships among qualifiers. In the example below, the subtype bound on `Q3` requires it to subsume both `Q1` and `Q2`:

```
def compose[Q1, Q2, Q3 >: Q1 with Q2](
  f1: (Int => Int) @mode[Q1], f2: (Int => Int) @mode[Q2]
): (Int => Int) @mode[Q3] = x => f2(f1(x))
```

It is also possible to avoid explicit qualifier constraints altogether and have a type-system extension that infers the constraints from the body and checks them at call sites.

Parametric polymorphism for 2nd-class values has been studied by Osvald et al. [45], and we can build on their $D^{1/2}_{<:}$-calculus, a variant of $D_{<:}$, which is a core calculus for a subset of Scala [3, 2, 48]. This system can encode $F_{<:}$ with 1st-class type values and path-dependent types. Their encoding carries over to our setting, resulting in an analogous $D^{1/2\,\hookleftarrow}_{<:}$-calculus [61]. This system abstracts over types and qualifiers separately.

Stack growth in generic code paths. Some operational aspects of storage-mode polymorphism require careful consideration. For a polymorphic return storage `@mode[Q]`, what code should a compiler generate? Keep or pop the function's stack frame when it returns? Consider a call tree of storage-polymorphic functions. If we instantiate `Q = Stack` we would expect the stack to grow, but what about `Q = Heap`?

There are three options: (1) no popping at all, (2) popping when polymorphic code returns to monomorphic code, and (3) popping throughout. A standard erasure implementation forces option (1), since only an instantiation `Q = Stack` can be assumed. Option (2) requires "compensation code" at call sites of parametric functions, delegating some of the stack-popping machinery to the caller instead of the callee. Option (3) can be achieved through monomorphization (i.e., compile-time specialization), or by tracking a runtime witness for each storage-mode parameter, making the popping decision via runtime dispatch.

```
// Storage-mode polymorphic vector addition (type signature):
def add[Q](v1: Vec[f32] @stack, v2: Vec[f32] @stack): Vec[f32] @mode[Q] =
  { val vout = new Vec[f32](v1.length); for (i <- v1.length) { vout(i) = v1(i) + v2(i) }; vout }
```

```
def addH(v1: Vec[f32] @stack,
         v2: Vec[f32] @stack) =
{
  markStack()
  val vout = allocH(v1.length)
  ...
  resetStack()
  vout // : Vec[f32] @heap
}
```

(a) Monomorphization (heap).

```
def addS(v1: Vec[f32] @stack,
         v2: Vec[f32] @stack) =
{
  // do not mark stack
  val vout = allocS(v1.length)
  ...
  // do not reset stack
  vout // : Vec[f32] @stack
}
```

(b) Monomorphization (stack).

```
def add[Q](v1: Vec[f32] @stack,
           v2: Vec[f32] @stack,
   implicit m: Storage[Q]) = {
  m.markPoly()
  val vout = m.alloc(v1.length)
  ...
  m.resetPoly()
  vout // : Vec[f32] @mode[Q]
}
```

(c) Dynamic dispatch (witness).

Figure 6 Achieving polymorphic allocation and stack growth behavior at runtime via (a,b) monomorphization, (c) dynamic dispatch with a storage-mode witness parameter. Exception return paths are elided in all three versions. Dynamic dispatch can be achieved using object or function indirection as shown, or, defunctionalized and inlined, as conditional dispatch on a single tag bit.

Monomorphization. This requires generating different versions of the parametric function for each distinct instantiation it is called with. Figures 6a, 6b show the specialized version of `add` for heap respectively stack storage mode. The code is ideal since there is no overhead and stack management code can be generated for option (3), at the cost of a blowup in code size.

Parametric stack/heap allocation with runtime witness. To avoid the issues with monomorphization one can instead create runtime witnesses for storage modes and pass those as extra arguments. While avoiding code duplication, some dispatch overhead is induced at runtime. We can also abstract over allocation policies in a storage-polymorphic way. Figure 6c abstracts over different allocators and whether the result buffer is on `add`'s stack frame or on the heap. Passing an allocator works for explicit data structures (e.g., arrays), but closure allocation requires compiler support.

6.3 Levels Beyond Stack and Heap: Tracking Effect Capabilities

We inherit Osvald et al's. [45] ability to include arbitrary lattices between 1st class (bottom) and 2nd class (top) via subtyping and phantom types [35]. Operationally, this type hierarchy can be interpreted as encoding *lattices of stacks*, for example:

```
type Heap  = Nothing   // 1st class
type Stack = Any       // 2nd class
type RS1   <: Stack    // phantom types
type RS2   <: Stack
type RH    <: RS1
```

Closures at a given level ℓ close over values at the same or a descendant level, and popping the ℓ stack pops all its descendant stacks. It would be an interesting extension to map individual storage levels to separate physical runtime stacks. But even in the absence of such runtime support, there are interesting use cases of fine-grained lattices for structuring application code, including user-managed stack/heap region allocation policies.

Beyond memory management, our $\lambda^{1/2}_{\hookleftarrow}$-calculus supports programming with capabilities, and forms of lightweight effect polymorphism as proposed by [45], such as the familiar `withFile` example in Figure 7. Our system is strictly more expressive, as we support returning

```
// Capabilities: 1 <: R <: RW <: 2
type CanR  = CanRead  @mode[R]
type CanRW = CanWrite @mode[RW]

// File I/O:
trait File(val path: String) {
  def read(implicit c: CanR): Byte
  def write(implicit c: CanRW): Unit
}
```

```
def withFile[U](...)(f: (File @mode[R] => U) @stack): U
def parReduce[U](...)(f: ((U, U) => U) @mode[R]): U
withFile("foo.txt") { f =>
  parReduce(1 to 1000) { (a, b) =>
    f.read()  // ok:    'R' can capture 'R'
    f.write() // error: 'R' can't capture 'RW'!
  }
  f.write()   // ok:    '2' can capture 'RW'
}
```

Figure 7 Tracking effect capabilities (example from [45]). Files support `read` and `write` methods requiring implicit capabilities, arranged in a privilege lattice; `withFile` provides a non-escaping file to a block which has full privileges. The nested `parReduce` block has only reading privileges.

2nd-class values. Lack of this ability impedes composability in practice, e.g., it is not possible to partially apply functions to capabilities. This issue inspired various solutions in related calculi [16, 43, 17, 15]. This paper shows that it can also be solved soundly and effectively with a small adjustment to the original system, namely allowing functions to return 2nd-class values. For example, the system now supports lazy collections in 2nd-class contexts:

```
withFile("foo.txt") { file =>
  val it = List(1,2,3,4).iterator.map(n => n + file.read().toInt) // iterator capturing file
  it.next() + it.next()  // ok (rejected in Osvald et al.'s system)
}
```

Finally, the example in Figure 7 also shows uses of nested scopes with *different* privilege levels, distinguishing read from write in the nested `parReduce` block, while maintaining the scoping guarantees of `withFile`. We have not seen evidence how other related works (e.g., capture types [15, 43] or reachability types [10]) can support this example with the same guarantees, so we pose it as a design challenge for works that seek to incorporate lightweight capability systems into realistic languages.

6.4 Stack References in Mutable Data Structures

One restriction we inherit from Osvald et al.'s system [45] is that 2nd-class values cannot be stored in mutable references/variables. Clearly, we do not want to store a stack reference in a heap-allocated mutable variable. But what about on-stack mutable variables? We have to be careful, since a reference could escape and be used after the containing stack frame is deallocated, e.g., if it was assigned to a variable further up the stack. However, assigning to on-stack variables is safe as long as the right-hand side was pushed on the stack before the variable. The key safety invariant is that stack references may only point upward, not downward. While a full solution for comparing lifetimes of variables seems possible using ownership [31, 59, 18, 41] or reachability types [10], this would add significant complexity. Still, partial solutions are possible using local reasoning on a best-effort basis, e.g., permitting assignments to variables within the same function.

Example: on-stack coroutines. Figure 8 shows a lowering transformation of a high-level coroutines API (adapted from [54]) to a stack-based implementation without heap allocations. To the left is the initial program in direct style, where a `decoder` continuously decompresses and feeds a character stream to a `parser` coroutine, encoded by delimited control operators `shift/reset` [21]. The `getChar` function for requesting the next character suspends the parser and transfers control back to the decoder, storing the parser's continuation in a shared mutable reference `emit`, which is invoked by the decoder once the next character is available. The shared state and the reified continuation should move from the heap onto the stack.

```
// Produce a stream of characters:
def decoder() {
  while (true) {
    var c = getRawChar()
    if (c == EOF) break
    if (c == 0xFF) { // decompress
      var len = getRawChar()
      c = getRawChar()
      while (len > 0)
        emit(c) // transfer control to parser
      len -= 1
    } else
      emit(c)    // transfer control to parser
  }
  emit(EOF)      // transfer control to parser
}
// Consume characters:
def parser(): Unit = reset {
  var c: Char = '_'
  while (c != EOF) { ①
    c = getchar()
    if (c != EOF) {
      if (c.isLetter) {
        do { ②
          addToToken(c)
          // transfer control to decoder
          c = getChar()
        } while (c.isLetter)
        gotToken(WORD)
      }
      addToToken(c); gotToken(PUNCT)
    }
  }
}
parser(); decoder()
```

```
// Read a character from stdin:
def getRawChar() = StdIn.readChar()
// Store parser continuation:
@stack private var emit: Char => Unit = _
// Suspend and wait for decoder:
def getChar() = shift {
  (k: (Char => Unit) @stack) =>
    emit = k
}
// After inlining & selective CPS transformation:
def parser(): Unit @stack = {
  var c: Char = '_'
  var f1: (Char => Unit) @stack = null
  var f2: (Char => Unit) @stack = null
  def doWhileGetChar(c1: Char) = { ②
    c = c1; if (c.isLetter) dowhile()
  }
  def doWhile(): Unit =
    { addToToken(c); emit = f1 }
  def outWhileGetChar(c1: Char) = { ①
    c = c1
    if (c != EOF) {
      if (c.isLetter) {
        doWhile(); gotToken(WORD)
      }
      addToToken(c)
      gotToken(PUNCT)
    }
    outWhile()
  }
  def outWhile(): Unit = if (c != EOF) emit = f2
  f1 = x => doWhileGetChar(x)
  f2 = x => outWhileGetChar(x)
  outWhile()
}
```

Figure 8 On-stack coroutines example. Left: read, decompress, and parse data from a stream using coroutines `parser` and `decoder` (adapted from [54]). Right: transformation of `parser` into mutually recursive functions with stack-allocated closures. The red and blue parts belong to the transformed outer (①) and inner loop (②), respectively.

A key question is how to track the lifetimes of the continuations. We generally do not know the dynamic extent and hence would require storage on the heap. However, in this case we know the lifetime of the `emit` variable and may thus store the continuations on the stack.

The final concern is avoiding stack overflows from continued invocations of `emit`, and indeed it is possible to let this program run in *constant* stack space. This requires a few standard transformations (Figure 8, right), i.e., inlining `getchar`, compiling the nested loops (① and ②) into mutually tail-recursive functions, and a selective CPS conversion for `shift/reset` using Cong et al.'s [19] technique based on 2nd-class functions.

6.5 Use-Site Driven Inference of Storage Modes

Passing a function expression to a higher-order function that expects a (potentially) stack-allocated closure will cause the closure to be indeed allocated on the stack:

```
map(list, v => v + 10)              // stack allocated
```

However, pulling out the function into a separate definition will cause it to be heap allocated:

```
val f = (v: Int) => v + 10          // heap allocated
map(list, f)
```

Unless the definition is marked explicitly for stack allocation:

```
@stack val f = (v: Int) => v + 10   // stack allocated
map(list, f)
```

```
// Example program:
fn f<F: FnOnce() -> i32>(g: F) {
  g();
}
let (t, mut s) = (1, 42);
f(|| { s += &t; s });

// Program after closure conversion:
fn f(g: Anon1) { g.call_once(()); }
let (t, mut s) = (1, 42);
f(Anon1 {s: &mut s, t: &t});
```

```
// Closure representation:
struct Anon1<'a>{
  s: &'a mut i32,
  t: &'a i32
}
impl<'a> FnOnce<()> for Anon1<'a> {
  type Output = i32; extern "rust-call"
  fn call_once(self, _unit: ()) -> i32 {
    *self.s += self.t;
    *self.s
} }
```

Figure 9 Closure conversion in Rust.

It is easy to propagate storage-mode information from uses to definitions in a local scope, and automatically convert local declarations to stack allocation that would otherwise default to heap allocation, if they are never used in a truly 1st-class way. This analysis can be implemented in a single pass, without fixpoint computation, if the definitions are already well-typed under the more general mode.

6.6 Function vs. Block Scope as Retention Boundary

For the most part of this paper, we have focused on functions as one particular control-flow construct, but it is also worth considering how stack growth and reclamation should behave in other block-scoped constructs, including loops.

Naively extending the lifetime of stack values to the function scope instead of the closest surrounding block scope would lead to exactly the same behavior as the `alloca` intrinsic in C. Notably, this would inhibit allocating stack values inside of a loop since it would quickly overflow the stack (left):

```
while (i < n) {
  @stack val tmp = new Vec[f32](10)
  use(tmp)
}
```

```
@inline def block(f: () => Unit) = f()
while (i < n) {
  block { ... }
}
```

Even without specific compiler support, it is possible to overcome this issue directly at the user level by using higher-order functions to denote block structure [32] (right). A block is simply a function that accepts a closure that is immediately applied.

Since the return type of `f` is `@heap` qualified, its stack-frame will be popped at the end of its body, making the program above run in constant stack space. Clearly, an equivalent transformation can be easily realized inside a compiler just as well.

6.7 Stack Allocation for Closures and Other Anonymous Structures

A crucial application of storage mode qualifiers is in the context of closures and more generally in returning anonymous structures implementing a specific trait. In both cases, the absence of a concrete type at the call-site inhibits pre-allocating space on the caller's stack frame for the returned value. In this section, we investigate the challenges of compiling traits/interfaces for stack allocations in modern languages with stack environments like Rust.

Closure expansion in Rust. Figure 9 shows an example program in Rust, where a function `f` calls its closure parameter `g`, along with the program's closure conversion. The concrete argument for `f` is an anonymous closure that captures two stack-allocated variables by reference. First, note that `f` is monomorphized for the specific closure type which will be passed by value. Second, the closure desugars into the struct `Anon1`, where its fields represent the captured environment, and the method `call_once` represents the closure's body.

```
// OK: returned stack-allocated closure
fn captureByV<F: Fn(i32) -> i32>(f: F)
  -> impl Fn(i32) -> i32 {
    move |v| f(v) + 10
}
// Error: the returned closure is a union
fn cond(a: i32, b: i32) -> impl Fn(i32) -> i32 {
  if (b != 0) { |v| a * v + b }
  else { |v| a * v }
}
```

```
// Error: impl in nested position ①
fn spicy_curry() -> impl Fn(i32) ->
  (impl① Fn(i32) -> i32) {
    |a| move |b| a + b
}
// Error : recursive closure requires boxing
fn omega(i: i32) -> impl Fn(i32) -> i32 {
  let res = omega(i - 1);
  move |v| res(v) + 1
}
```

Figure 10 Valid and invalid returns of stack-allocated closures in Rust.

Limitations. To support stack allocation of closure objects (`structs`) the Rust compiler must be able to compute the size of the closure environment statically. In Figure 9, this is made possible by monomorphization, e.g., `Anon1` has the size of two `i32` references (the captured variables). This treatment of closures restricts their uses in return position. When generating code for a call to a function that returns a stack-allocated closure, the compiler has to reserve space for the returned closure in the caller's frame. However, it cannot know the size, because function types do not convey anything about the result's captured environment.

To mitigate this issue, Rust introduced abstract return types, which allow returning anonymous stack-allocated `structs` (e.g., `captureByV` in Figure 10). However, this feature comes with its own limitations – only a value of a single concrete type can be returned. The concrete return type is only superficially anonymous: the compiler tracks it to compute the required size for the stack allocation. This disallows certain programs (Figure 10), where the returned closure is data dependent (`cond`), or when a stack-allocated closure is returned from another closure (`spicy_curry`). The first problem can be addressed by pessimistically preallocating the size of the maximum closure that will be returned. The second one cannot be solved – the Rust compiler relies on the fact that it can always identify the concrete underlying type for any `impl` type. Allowing it in nested positions breaks this property. Another problem is that captured environments can be recursive (`omega` in Figure 10) in which case Rust requires a boxed representation on the heap.

Storage-modes solution. All the examples in Figure 10 are supported by our system. For example, the `spicy_curry` definition can be implemented as

```
val curry: (i32 => (i32 => i32) @stack①) @stack = a => b => a + b
```

The `@stack` qualifier in the return type of a closure (①) renders it impossible to identify the size of the returned value at compile time, unless a whole-program analysis is performed. This is not at all required with storage modes, e.g., the following definition allows both separate compilation and partial applications yielding closures of unknown size:

```
// Module 1:
def f(g: (i32 => (i32 => i32)@stack)): i32 =
  // Closure of unknown size:
  val g' = g(10)
  g'(2)

// Module 2:
f(v1 => v2 => v1 + v2) // ok
```

```
// Module 3:
def incBy(v: i32): (i32 => i32) @stack =
  if (v == 0) { x => x }
  else { // recursive closure
    val rec = incBy(v - 1)
    { x => rec(x) + 1 }
  }
f(incBy(_)) // ok
```

Closures are not the only instance where storage modes are effective. Another one is in the context of lazy iterators, where a sequence of operations such as `fold` ∘ `map` would generate intermediate iterators which will eventually be consumed. The Rust implementation of such combinators relies on returning concrete monomorphized structures so that they can be stack allocated. Storage-mode qualifiers achieve this in a more straightforward manner:

```
class Linear(s: (Int, Int)) extends Layer {
  val W = Tensor.randn(s._1 :: s._2 :: Nil)                    // weights
  val b = Tensor.randn(s._1 :: Nil)                            // biases
  type TensorCPS = (Tensor @stack => Unit) @stack ③ => Unit
  def apply(x: Tensor @stack ①): TensorCPS @stack ② = { k =>
    val h = Tensor.matmuladd(W, x, b)                          // forward pass
    k(h)                                                       // ... continuation ...
    Tensor.∇matmuladd(h.∇, W, x, b)                            // backward pass
  }
}
```

Figure 11 Definition of a linear layer where h is allocated on the stack by `matmuladd` and is freed on returning along the backward pass.

```
def map[A, B](it: Iter[A] @stack, f: (A => B) @stack): Iter[B] @stack =
  new Iter[B] { def hasNext = it.hasNext;  def next() = f(it.next()) }
```

Finally, the stack semantics introduced by storage-mode qualifiers enables compilers to avoid treating `alloca` as an intrinsic and define it as a library function:

```
def alloca[T](n: Int): Array[T] @stack
```

7 Evaluation

In this section, we show how storage modes can improve memory management of larger programs. We first describe two case studies, differentiable programming and parser combinators, and then conclude with an in-depth evaluation of storage-mode-annotated programs.

7.1 Case Study: Differentiable Programming

While storage modes are widely applicable for returning short-lived variable-size data, they are also useful for arena-style memory management. For example, computing gradients in differentiable programming requires keeping the intermediate tensors produced by operations in the forward pass of the dataflow graph until the operation node is traversed in the backward pass. The forward and backward pass are executed multiple times on different input tensors. Peak memory usage can be statically upper-bounded. However, the dataflow graph can be data dependent. Therefore, storage for intermediate tensors cannot be statically resolved. Instead, it has to be managed dynamically. Deep learning frameworks handle this by building a separate allocator with reference counting for managing the allocations in the graph.

Here we take a different approach, and implement a differentiable program using delimited continuations à la Wang et al. [58, 57]. Every primitive operation of the model (e.g., linear operators) takes a continuation denoting the rest of the forward and backward pass of the model. Figure 11 shows the definition of a linear layer that computes $l(x) = W_l \cdot x + b_l$. Ignoring the `@stack` annotations, applying the linear layer first computes $W_l \cdot x + b_l$ through `matmuladd` producing the tensor h. It then calls the continuation which will eventually update the gradient field $h.\nabla$ of h and finally computes and accumulates the gradients for W and b.

Since the argument of function `apply` is annotated with `@stack` (①), its returned function of type `TensorCPS` must also be `@stack` (②). It cannot capture the argument otherwise. We also note that since the continuation `k` is used in a non-escaping fashion, it can be annotated as `@stack` (③). Finally, the tensor returned by `matmuladd` is also stack allocated. The full type of the returned function is `(((Tensor @stack => Unit)@stack)=> Unit)@stack`. Since its return type is not `@stack` annotated but its body calls a function that returns a `@stack`-annotated

```
class MLP(shapes: List[(Int, Int)]) {
  val (l1, l2, l3) = ...
  def apply(x: Tensor) =
    for {
      z1 <- l1(x)
      h1 <- σ(z1)
      z2 <- l2(h1)
      h2 <- σ(z2)
      o  <- l3(h2)
    }
    yield o
}
```

```
val mlp = MLP(...)
for (i <- 0 until epochs) {
  for ((x, y) <- data) {
    grad { x =>
      for {
        o <- mlp(x)
        ℓ <- loss(y, o)
      } yield ℓ
    }(x)
    pars.foreach(optimize(_, η))
  }
}
```

Figure 12 Fully differentiable program (multi-layer perceptron, MLP) with training loop. Intermediate values are stored on the stack and de-allocated eagerly during the backward pass. `MLP` is visualized in a dataflow graph.

```
// Infix stack-function arrow:
type =@>[-A,+B] = (A => B) @stack

// On-stack parser type:
type SP[+U] = Parser[U] @stack

trait Result[+T] {
  def map[U](f: T =@> U): Result[U]
  def andThen[U](f: T =@> SP[U]): Result[U]
  def append[U >: T](alt: => Result[U]): Result[U]
}
```

```
@stack
class Parser[+T](val f: Input =@> Result[T]) {
  def flatMap[U](g: T =@> SP[U]): SP[U] =
    new Parser(f(_).andThen(g))
  def map[U](g: T =@> U): SP[U] =
    new Parser(f(_).map(g))
  def |[U >: T](p: => SP[U]): SP[U] =
    new Parser(in => f(in).append(p(in)))
  def ~[U](p: => SP[U]): SP[(T, U)] =
    this.flatMap(a => p.map(b => (a, b)))
}
```

Figure 13 On-stack parser combinators. Intermediate parser objects are short-lived. Making them stack-allocated reduces ephemeral heap allocations and fragmentation.

result, the compiler will insert marking and release instructions. This results in *every stack allocation happening in the execution of this function to be deallocated when returning*. This is safe because the continuation `k` uses the tensor `h` in a non-escaping fashion.

Figure 12 shows a fully differentiable program in our model. The `for` expressions are Scala's syntax for monadic comprehensions. Every temporary allocation performed by `l`i or σ is discarded after the backward operation of the respective layer as shown for the `Linear` layer before. The dataflow graph in the center depicts the `apply` function of the `MLP` class to the left. We represent continuations as dashed gray boxes. To the right is a program training an `MLP` instance. The program achieves arena-style memory management only through stack allocations and storage-mode qualifiers that ensure validity of references. The program is lightweight in annotations since those can be inferred given the definitions in Figure 11.

7.2 Case Study: Parser Combinators

The differentiable programming case study (Section 7.1) allocates most of the data on the stack (e.g., intermediate tensors). Yet, storage modes are also beneficial for interleaving stack and heap allocation. An interesting example where this occurs is parser combinators.

Consider the definition of `Parser` in Figure 13. A parser produces a `Result[U]` which can be either `Failure` or `Success`. Parsers are built by combining other "base" parsers through combinators such as alternation `|` and sequencing `~`. For instance, the latter produces a parser that first applies a parser `f` on the input and then applies another parser `p` on the rest of the input producing two outputs. Note that the combinators' arguments are by-name which enables recursive parsers.

All `Parser` instantiations are allocated on the stack. This means that values closed over by parsers can also be marked as `@stack` (e.g., `g` in `flatMap`). To understand how the `@stack` annotations affect the computation, consider the following example that parses the strings matching the regular expression `(ab)*`: `def (ab)* = (lit("ab") ~ (ab)*) | lit("")`.

The `lit` combinator allocates a parser that recognizes the argument string. The recursive call is passed by name as argument to the `~` combinator, meaning that it will produce an allocation only when forced in the body of the newly built parser. Since parser functions have type `(Input => Result[T])`, the allocations of `Parser` happening on the stack will be reclaimed on return. During the evaluation of `(ab)*` on an input string, each recursive call will push a new parser object on the stack. Once the full input has been consumed, each stack-allocated parser object will be popped during the return path of the recursive calls.

Annotating `Parser` objects as `@stack` is beneficial in two ways: (1) it reduces the amount of ephemeral allocations on the heap, reducing overhead induced by collections, and (2) it reduces heap fragmentation. Running a parser results in alternating allocations of long-lived objects (the parsed results) and short-lived intermediate `Parser` objects. Many runtimes (e.g., Java and OCaml) handle this with generational garbage collection, allocating new objects in "young" regions and moving them to "old" regions once they stay reachable for long enough. In contrast, storage modes reduce heap fragmentation without relying on any assumption about the underlying heap memory management.

7.3 Performance Evaluation

We evaluate our storage-mode implementation in Scala Native (Section 5) on a range of benchmarks:

1. **filter**: This benchmark executes the code described in Section 2.2, which allocates the filtered list on the stack. The list size is 50 and 1/3 of the items are kept.
2. **flatten**: This benchmark flattens a binary tree of height 5 (32 nodes) by first building a tree of closures on the stack and then builds the list on the heap.
3. **curry**: A loop that curries a function.
4. **parser**: On-stack parser combinators, where parsed return values are heap allocated.
5. **graph**: Simulates insertions and deletions of edges and nodes on a graph. Insertions happen at a higher probability (55%) than deletions. At each insertion, the maximum distance between any 2 adjacent nodes is computed for the updated node. The graph is represented through adjacency lists on the heap. The maximum distance is computed using nested stack-allocated iterators.
6. **mlp** is the code shown in Section 7.1; most of the structures are stack allocated.

We compare each program annotated with `@stack` to its counterpart without `@stack` annotations. Everything else (compiler options, number of runs, etc.) remains unchanged. Benchmarks are compiled with Scala Native [50] generating LLVM byte code, which is then compiled through LLVM v10.0 with -O2. The benchmarks were run on a Intel Core i5-6300HQ CPU @ 2.30GHz on Ubuntu 20.04 LTS, Linux Kernel v5.11.0. All benchmarks stress the underlying memory management subsystem.

End-to-end runtime improvement. The table in Figure 14 shows averages and standard deviations of run times and the speedup for each benchmark. Each benchmark starts from a clean state and is run in a hot loop. The storage-modes version of each benchmark outperforms the unannotated version, and we observe average speedups of up to 25%.

	AVG	AVG (`@stack`)	STD	STD (`@stack`)	AVG SPEEDUP
flatten (μs)	1.57	1.31	1.55	1.57	1.20x
mlp (ms)	859.27	702.12	2.99	4.58	1.22x
curry (μs)	22.14	20.24	6.82	4.94	1.09x
parser (μs)	707.25	563.68	59.94	79.95	1.25x
graph (ms)	32.66	28.14	3.39	2.10	1.16x
filter (μs)	86.43	77.78	16.48	14.29	1.11x

Figure 14 Run time comparison of annotated vs. unannotated programs. The chart shows the overhead of memory management as percentage of the total running time: "alloc" represents heap management in the unannotated version, and "heap + stack" represents heap and shadow stack management for the annotated version.

Table 1 Memory management evaluation. Columns show the number of GC executions, maximum shadow stack depth in bytes, time spent in the marking phase and sweeping phase.

	MODE	COLLECTIONS	STACK DEPTH (BYTES)	MARK MEDIAN (μs)	MARK TOTAL (ms)	SWEEP MEDIAN (μs)	SWEEP TOTAL (ms)
flatten	-	11126	0	16.79	200.82	15.33	177.20
	`@stack`	4950	2520	17.06	91.32	15.78	81.45
mlp	-	363	0	40.27	15.85	151.56	58.06
	`@stack`	116	18944	42.48	5.33	143.35	17.94
curry	-	1128	0	14.23	18.16	15.41	19.11
	`@stack`	342	40	14.76	5.87	15.04	5.62
graph	-	22	0	723.46	15.03	840.10	14.78
	`@stack`	14	416	1590.99	20.51	515.47	6.69
filter	-	967	0	16.65	18.16	16.29	17.01
	`@stack`	-	-	-	-	-	-
parser	-	51	0	21.36	1.15	15.66	0.88
	`@stack`	18	28448	21.58	0.49	16.40	0.33

Memory management overhead. The bar chart in Figure 14 shows the overhead percentage of memory management as computed through `perf v5.11.22` [28] with frequency `-F 30000`. Such a percentage is the number of times a memory management function activation record was alive during execution over the total samples of the stack. Bars on the left are percentages for the unannotated program, using the heap for most structures while bars on the right are the ones of the annotated program. Both percentages are scaled with respect to the running time of the unannotated program. The bars on the right of each benchmark are partitioned in the overhead for managing the heap (bottom) and the overhead for managing the stack (top). The size of these partitioned bars are also proxy metrics for the amount of allocations that happen on the stack for the annotated program. Overall, storage modes reduce the memory management overhead, since garbage collection is reduced.

Garbage collection behavior. Table 1 shows important statistics about the underlying memory management subsystem. First, we note that the number of times a collection is triggered in each benchmark is dramatically reduced. The median time for the marking and sweeping phases of each collection is shown in the table. The total time spent in marking and sweeping is more than halved, producing the performance improvements shown in Table 1.

An outlier is the increase in median time for marking in the graph benchmark. This is because a collection is triggered only after the heap is at maximum capacity. For the `@stack` version, this happens later in time, since ephemeral allocations due to iterators are stack allocated. At this point, the heap-allocated graph is larger and marking is more costly.

Currently, shadow-stack allocations are implemented as calls to an external function. It would be possible to further improve performance by optimization passes that exploit the non-escaping behavior imposed by the type system. Also, the marking phase of the GC still has to traverse the shadow stack (similarly to the call stack) to find heap roots. Hence, languages without GC can benefit even more from storage modes. For example, Swift uses reference counting, and making closure arguments non-escaping yields major performance improvements [53], which prompted the designers to make closure arguments non-escaping by default. Our storage modes provide strictly more benefits: not only can arguments be non-escaping, but so can be function return values, and they may be of variable size.

8 Related Work

Our work is inspired by Osvald et al. [45], who argued in favor of reintroducing 2nd-class values in modern programming languages. Instead of relying on escape analysis, they introduce $\lambda^{1/2}$, a language with qualified types where variables can be restricted to 2nd-class status, ensuring non-escaping behavior. The typing rules ensure that 2nd-class values follow a *strict* stack discipline and have been employed to model scoped capabilities [45, 46, 17, 43], in compiler intermediate representations for efficient compilation [19], and as compiler directives to improve stack allocation of closures [6]. We remove the limitation of their type system by enabling the return of 2nd-class values, which increases their composability, e.g., allowing currying over 2nd-class values. To achieve that, we propose a provably sound, relaxed stack semantics. Finally, instead of using the type system only for capability checking on the JVM version of Scala, we employ it for stack allocation on Scala Native [50] and provide an evaluation regarding the performance benefits of the system.

The problem of implementing programming language environments through a stack or a heap dates back to the Algol and LISP communities in the 70s. The LISP community named the problem as the "Funarg problem" [39] and the Algol community named it "Retention vs. deletion strategy" [12]. These works investigated the problem of escaping stack references in the presence of higher-order functions. Fischer [24] proved that the retention and deletion strategies were equally expressive by providing a transformation later recognized as one of the discoveries of CPS [47]. Our work combines these ideas, dealing with stack references through storage modes and returning variable-size data by implementing a delayed "deletion" strategy inspired by the corresponding selective CPS transformation [19].

The implementation of the shadow stack resembles Obstacks [26], which are "bump" memory allocators where allocation and deallocation is managed by pointer adjustments. The Obstacks API can be used as an architecture-independent lowering target to implement shadow stacks, but it does not come with static guarantees of its own.

Banerjee and Schmidt [9] designed a static criterion to approximate the runtime shape of the environment and determine whether a term could be evaluated using a single-stack environment. Their work extends the "simple expression" work of Georgeff [27]. Appel and Shao [5] argued in favor of fully allocating activation records on the heap.

Our approach is the first to enable stack allocation of variable-size data in direct style and can be retrofitted into language implementations relying on stack environments. The effectiveness of stack allocation has been investigated, among others, by Baker in the implementation of Scheme [8]. Efforts to add stack allocation are currently underway in many languages, including C♯ [36], Swift [53], and OCaml [42, 30].

Closely related to our work are region-based systems as proposed by Tofte and Talpin [55, 56]. While storage modes can be translated to a region calculus, our implementation relies on a simpler type system and does not require designing the language around region-based memory

management. Compared to region-based memory management with explicit annotations, our solution is lightweight in terms of annotations. To ameliorate the annotation burden, region calculi are equipped with region inference. Storage modes are intentionally explicit to provide the programmer control over the stack, but also integrate well with type inference in languages that support it. Our work shows benefits in terms of performance, especially for compiled languages that automatically manage memory through garbage collection or reference counting. Nonetheless, storage modes are also compatible with type systems based on principles of ownership and borrowing, such as Rust's [31, 59], and can be beneficial to support variable-size data when compiling for embedded systems without support for dynamic heap allocations.

9 Conclusions

This paper addresses the problem of returning variable-size data in languages with stack environments. Our approach relies on storage modes, which are lightweight type annotations that guide stack allocations and de-allocations. The evaluation of storage modes as implemented in the Scala-Native compiler shows that our approach is beneficial for both reducing heap fragmentation, GC overhead and improving spatial locality. Storage modes can be implemented in high-level languages such as Swift and Scala to reduce heap pressure or in low-level languages such as Rust to promote uses of abstractions without paying the heap penalty.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, pages 8–19. ACM, 2003. `doi:10.1145/888251.888254`.
2. Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. `doi:10.1007/978-3-319-30936-1_14`.
3. Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *POPL*, pages 666–679. ACM, 2017. `doi:10.1145/3009837.3009866`.
4. Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, 1987. `doi:10.1016/0020-0190(87)90175-X`.
5. Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *J. Funct. Program.*, 6(1):47–74, 1996. `doi:10.1017/S095679680000157X`.
6. Apple Inc. Closures – The Swift Programming Language (Swift 5.4), May 2021. URL: `https://web.archive.org/web/20220501162412/https://docs.swift.org/swift-book/LanguageGuide/Closures.html`.
7. Kenichi Asai and Chihiro Uehara. Selective CPS transformation for shift and reset. In *PEPM*, pages 40–52. ACM, 2018. `doi:10.1145/3162069`.
8. Henry G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *ACM SIGPLAN Notices*, 30(9):17–20, September 1995. `doi:10.1145/214448.214454`.
9. Anindya Banerjee and David A. Schmidt. Stackability in the simply-typed call-by-value lambda calculus. *Sci. Comput. Program.*, 31(1):47–73, 1998. `doi:10.1016/S0167-6423(96)00040-8`.
10. Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021. `doi:10.1145/3485516`.

11 Friedrich L. Bauer. The cellar principle of state transition and storage allocation. *IEEE Ann. Hist. Comput.*, 12(1):41–49, 1990. `doi:10.1109/MAHC.1990.10004`.

12 Daniel M. Berry. Block structure: Retention or deletion? (extended abstract). In *STOC*, pages 86–100. ACM, 1971. `doi:10.1145/800157.805041`.

13 Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log.*, 9(1):6, 2007. `doi:10.1145/1297658.1297664`.

14 Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, pages 22–32. ACM, 2008. `doi:10.1145/1375581.1375586`.

15 Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondrej Lhoták, and Martin Odersky. Tracking captured variables in types. *CoRR*, abs/2105.11896, 2021. `arXiv:2105.11896`.

16 Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–30, 2022. `doi:10.1145/3527320`.

17 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA):126:1–126:30, 2020. `doi:10.1145/3428194`.

18 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013. `doi:10.1007/978-3-642-36946-9_3`.

19 Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.*, 3(ICFP):79:1–79:28, 2019. `doi:10.1145/3341643`.

20 Olivier Danvy. Defunctionalized interpreters for programming languages. In *ICFP*, pages 131–142. ACM, 2008. `doi:10.1145/1411204.1411206`.

21 Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160. ACM, 1990. `doi:10.1145/91556.91622`.

22 Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *PPDP*, pages 162–174. ACM, 2001. `doi:10.1145/773184.773202`.

23 Edsger W. Dijkstra. Recursive Programming. *Numerische Mathematik*, 2(1):312–318, December 1960. `doi:10.1007/BF01386232`.

24 Michael J. Fischer. Lambda calculus schemata. In *Proving Assertions About Programs*, pages 104–109. ACM, 1972. `doi:10.1145/800235.807077`.

25 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203. ACM, 1999. `doi:10.1145/301618.301665`.

26 Free Software Foundation, Inc. The GNU C Library–Obstacks, 2003. URL: `https://web.archive.org/web/20220509075117/https://www.gnu.org/software/libc/manual/html_node/Obstacks.html/`.

27 Michael P. Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Trans. Program. Lang. Syst.*, 6(4):603–631, 1984. `doi:10.1145/1780.1803`.

28 Brendan Gregg. Linux Perf Examples, 2020. URL: `https://web.archive.org/web/20220509003430/https://www.brendangregg.com/perf.html`.

29 Sten Henriksson. A brief history of the stack. In *SIGCIS Workshop*, 2009.

30 Jane Street. Memory management, 2022. URL: `https://web.archive.org/web/20220401080322/https://signalsandthreads.com/memory-management/`.

31 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018. `doi:10.1145/3158154`.

32 Peter J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965. `doi:10.1145/363744.363749`.

33 James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, California Univ., Berkeley, Dept. of Electrical Engineering and Computer Sciences, May 1989.

34 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

35 Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122. ACM, 1999.

36 Microsoft. C♯ language reference – stackalloc expression, 2022. URL: https://web.archive.org/web/20220405070936/https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/stackalloc.

37 James S. Miller and Guillermo Juan Rozas. Garbage collection is fast, but a stack is faster. Technical report, Massachusetts Institute of Technology, 1994.

38 James H. Morris. A bonus from van Wijngaarden's device. *Commun. ACM*, 15(8):773, August 1972. doi:10.1145/361532.361558.

39 Joel Moses. The function of FUNCTION in LISP, or why the FUNARG problem should be called the environment problem. Technical report, Massachusetts Institute of Technology, USA, 1970. URL: http://hdl.handle.net/1721.1/5854.

40 Lasse R. Nielsen. A selective CPS transformation. In *MFPS*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 311–331. Elsevier, 2001. doi:10.1016/S1571-0661(04)80969-1.

41 James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998. doi:10.1007/BFb0054091.

42 OCaml Community. OCaml discourse: Add support for stack allocation, 2021. URL: https://web.archive.org/web/20210125181231/https://discuss.ocaml.org/t/add-support-for-stack-allocation/7039.

43 Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. Safer exceptions for scala. In *SCALA@SPLASH*, pages 1–11. ACM, 2021. doi:10.1145/3486610.3486893.

44 Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *POPL*, pages 41–53. ACM, 2001. doi:10.1145/360204.360207.

45 Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*, pages 234–251. ACM, 2016. doi:10.1145/2983990.2984009.

46 Leo Osvald and Tiark Rompf. Rust-like borrowing with 2nd-class values (short paper). In *SCALA@SPLASH*, pages 13–17. ACM, 2017. doi:10.1145/3136000.3136010.

47 John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3):233–247, November 1993. doi:10.1007/BF01019459.

48 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *OOPSLA*, pages 624–641. ACM, 2016. doi:10.1145/2983990.2984008.

49 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*, pages 317–328. ACM, 2009. doi:10.1145/1596550.1596596.

50 Denys Shabalin. Scala native, 2015. URL: https://web.archive.org/web/20220318165107/https://scala-native.readthedocs.io/en/latest/.

51 Amir Shaikhha, Andrew W. Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In *FHPC@ICFP*, pages 12–23. ACM, 2017. doi:10.1145/3122948.3122949.

52 Jeremy Siek. Type Safety in Three Easy Lemmas, 2013. URL: https://web.archive.org/web/20220308042857/https://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.

53 Swift Community. Use stack allocation for Swift closures #21933. URL: https://github.com/apple/swift/pull/21933/#issuecomment-454980737.

54 Simon Tatham. Coroutines in C, 2000. URL: https://web.archive.org/web/20220428071140/https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html.

55 Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *POPL*, pages 188–201. ACM Press, 1994. doi:10.1145/174675.177855.

56 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

57 Fei Wang, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *NeurIPS*, pages 10201–10212, 2018.

58 Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, 2019. doi:10.1145/3341700.

59 Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The Essence of Rust. *arXiv:1903.00982 [cs]*, August 2020. arXiv:1903.00982.

60 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

61 Anxhelo Xhebraj, Oliver Bračevac, Guannan Wei, and Tiark Rompf. What if we don't pop the stack? The return of 2nd-class values (extended version). Technical report, Purdue University, 2022.

Maniposynth: Bimodal Tangible Functional Programming

Brian Hempel ✉ 🏠
University of Chicago, IL, USA

Ravi Chugh ✉ 🏠
University of Chicago, IL, USA

Abstract

Traditionally, writing code is a non-graphical, abstract, and linear process. Not everyone is comfortable with this way of thinking at all times. Can programming be transformed into a graphical, concrete, non-linear activity? While nodes-and-wires [71] and blocks-based [3] programming environments do leverage graphical direct manipulation, users perform their manipulations on abstract syntax tree elements, which are still abstract. Is it possible to be more concrete – could users instead directly manipulate live program values to create their program?

We present a system, MANIPOSYNTH, that reimagines functional programming as a non-linear workflow where program expressions are spread on a 2D canvas. The live results of those expressions are continuously displayed and available for direct manipulation. The non-linear canvas liberates users to work out-of-order, and the live values can be interacted with via drag-and-drop. Incomplete programs are gracefully handled via hole expressions, which allow MANIPOSYNTH to offer program synthesis. Throughout the workflow, the program is valid OCaml code which the user may inspect and edit in their preferred text editor at any time.

With MANIPOSYNTH's direct manipulation features, we created 38 programs drawn from a functional data structures course. We additionally hired two professional OCaml developers to implement a subset of these programs. We report on these experiences and discuss to what degree MANIPOSYNTH meets its goals of providing a non-linear, concrete, graphical programming workflow.

2012 ACM Subject Classification Software and its engineering → Integrated and visual development environments; Software and its engineering → Visual languages; Software and its engineering → Programming by example; Human-centered computing → Human computer interaction (HCI)

Keywords and phrases direct manipulation, tangible programming, programming user interfaces

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.16

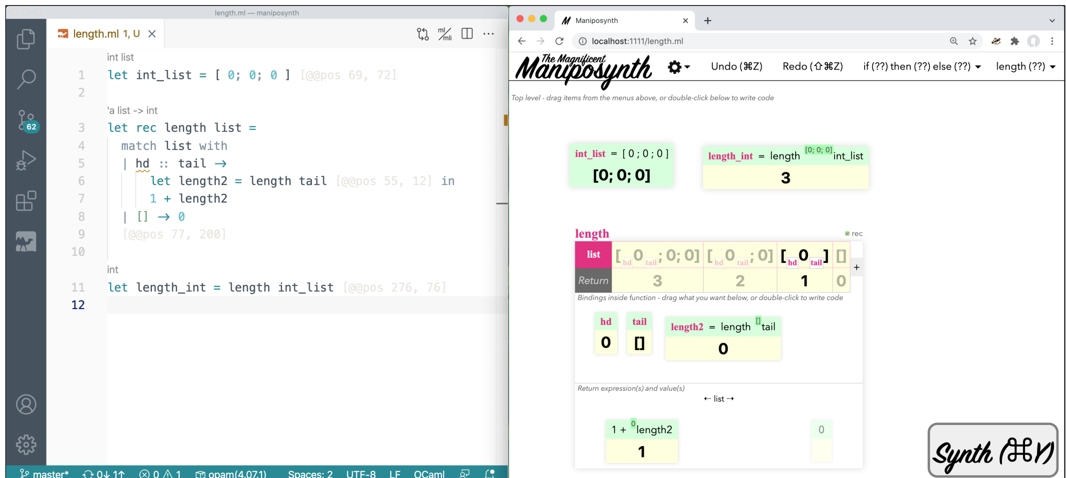


Figure 1 A list `length` function implemented in MANIPOSYNTH.

© Brian Hempel and Ravi Chugh;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 16; pp. 16:1–16:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work was supported by U.S. NSF grant *Direct Manipulation Programming Systems* (CCF-1651794).

Acknowledgements We extend our gratitude to Justin Lubin for advice (so far ignored) about caching during synthesis, Byron Zhang for feedback on the presentation, Matt Teichman for providing hardware, Kartik Singhal for technical support, and the user study participants for their great patience with bugs and for the invaluable feedback they provided.

1 Introduction

Graphical, direct manipulation interfaces [66] are the paradigm most users are familiar with when they operate computers. Graphical interfaces are powerful and vastly extend the reach of computing. Nevertheless, the most powerful computer activity – programming – has proven resistant to manifestation in a graphical, direct manipulation form. Most programming is primarily a text-only activity. Can general-purpose programming be reimagined in a graphical, direct manipulation interface? Experts might find productivity gains and novices might find a more approachable environment to accomplish their goals.

Most existing graphical programming approaches present the abstract syntax tree (AST) elements as items to be manipulated with the cursor. Nodes-and-wires programming environments [71], such as LabVIEW [55], present program expressions as boxes whose inputs and outputs are connected by wires. Blocks-based environments, such as Scratch [62], present the program expressions as puzzle pieces that snap together. And structure editors, such as Barista [42], allow certain manipulations of program expressions as structured entities rather than as a naive string of text. All these approaches center the AST as the object of interaction. Even more concrete than AST elements, however, are the *values* a program produces during execution. Humans are concrete thinkers before we are abstract thinkers, and teachers know that the best way to explain is through examples. So, is there a way to write programs via direct manipulation on *values* instead of on AST elements?

The Eros environment [19] demonstrated a compelling answer to this question. Eros reimagined the programming space not as a program in text (as in traditional coding) nor as a draftsman's drawing of operations connected by wires (as in nodes-and-wires programming), but as a 2D canvas of malleable values. These *tangible values (TVs)* were primarily partially applied functions, rendered with (graphically editable) example arguments for their unapplied inputs, with the corresponding example output displayed below. The user could select the output of one TV, the input of another TV (of corresponding type), and compose the two together into a new TV.

Eros highlighted that *non-linear editing* and pure functional programming are complementary. Without side effects, the order of computation is negligible. The user may gather the parts they need in any order and worry later about how to assemble them. Alas, the standard practice of writing functional programs as linear, textual code obscures this opportunity for non-linearity. Placing values on a 2D canvas instead highlights it.

Non-linearity matters. Not all humans are linear thinkers, and not even all programmers think linearly at all times. (How often are large blocks of code written top-to-bottom from scratch?) A non-linear environment can offer a creative space more inviting to folks whose standard workflow naturally entails concrete exploration rather than abstract planning.

While Eros highlighted how non-linear editing dovetails with pure functional programming, its mechanism for composing TVs may tip the balance too far from the abstract in favor of the concrete. Once a value has been composed, it obscures *how* it came to be. TVs are

labeled with a brief expression, but this one line is inadequate for any computation of modest size. Moreover, once composed, how does one change the computation that produced a TV? Value manipulation alone may be inadequate for carefully specifying abstract algorithms. Perhaps there is a middle ground that allows both non-linear, concrete direct manipulation on values *and* traditional editing of ordinary code. That middle ground is the subject of this paper. In particular, we seek to answer the question:

> *How can the approachability of non-linear direct manipulation on concrete values be melded with the time-proven flexibility of text-based coding?*

Design Goals. We aim to create a programming interface with the following properties:

(a) **Value-Centric.** Like Eros, and unlike most visual programming environments, we want values – not AST elements – to be centered in the display and, as much as possible, be the object of the user's direct manipulations.

(b) **Non-Linear.** To support non-linear thinkers and exploratory programming, we want to allow the user to gather the parts they need out of order, and compose them later.

(c) **Synthesis.** How to integrate recent advances in program synthesis into a practical workflow remains an open question. A value-centric interface is a natural environment to specify assertions on those displayed values and to fulfill those assertions with a synthesizer – we want to explore this.

(d) **Bimodal.** Ideally, a visual programming environment would not sacrifice the unique affordances of textual code – its concision and its amenability to an ecosystem of existing tooling (such as text editors, language servers, and version control). We want to offer a bimodal interface that simultaneously offers a non-linear graphical editing interface *alongside* a text-editable, traditional representation of the program's code.

Contributions. To show how value-centric non-linear editing can meld with traditional text-based programming, we implemented a value-centric, non-linear, bimodal programming environment with synthesis features called ***(The Magnificent) Maniposynth***. We demonstrate both how non-linear visual editing can integrate with linear code, as well as show novel editing features made possible by the value-centric display.

To gain an initial understanding of the system, we implemented an external corpus of 38 example programs. For additional insights, we conducted an in-depth exploratory study with two external professional functional programmers, whose feedback informed the evolution of Maniposynth. We describe their use of the tool and discuss additional observations through investigative lenses from the Cognitive Dimensions of Notation framework [25].

Section 2 introduces Maniposynth with a running example. Section 3 describes the technical implementation of the tool and the synthesizer. Section 4 presents insights from implementing a corpus of examples and the qualitative user study. Section 5 presents related work, and Section 6 discusses avenues for continued exploration.

2 Overview Example

To provide an overview of Maniposynth, we follow a fictional programmer named Baklava as she re-implements the list `length` function from scratch. Figure 1 shows the final result. A video of this example, as well as an artifact to follow along, are available online [32].

[
[0; 0; 0]
[0; 0]
[0]
[]

int_list = [0 ; 0 ; 0]
[0; 0; 0]

length_int = length [0; 0; 0] int_list
?

Figure 2 List literals in autocomplete menu.

Figure 3 Tangible values (TVs) for the example list binding and the example call to `length`.

Maniposynth is a locally running web application designed to be opened in a web browser alongside the user's preferred text editor. Baklava creates a blank text file named `length.ml` on her computer, starts Maniposynth in that directory, and navigates to `http://localhost:1111/length.ml` in her web browser. She positions her browser window side-by-side with Visual Studio Code [53] and is ready to begin.

2.1 List Length, Without Synthesis

To start, Maniposynth displays a blank white 2D canvas. Because Maniposynth is a live programming environment, Baklava starts by creating an example list so she can see the `length` operation on concrete data. Double-clicking on the canvas opens up a text box to add new code; Baklava does so and types an open bracket `[`. Because writing example data is common in Maniposynth, concrete literals up to a fixed size are offered as autocomplete options (auto-generated from the data constructors in scope, Figure 2). Baklava selects the list literal `[0; 0; 0]` from the autocomplete options and hits Enter.

In the code, a new let-binding for the list is inserted at the top-level of `length.ml` and automatically given the name `int_list`. On the canvas, this binding is represented as a box displaying (in clockwise order, Figure 3, left) the binding pattern (`int_list`), the binding expression (`[ 0 ; 0 ; 0 ]`), and the result value below (also `[ 0; 0; 0 ]`, but bigger). These three elements together in a box form a *tangible value* in Maniposynth. The box may be repositioned on the 2D canvas, and the coordinates of the position are stored in the code as an AST attribute annotation on the binding, written `[@@pos 152, 49]` in the code. Arbitrary attribute annotations are supported by the standard OCaml AST which allow these properties to be preserved across program transformations. Baklava has installed a VS Code plugin to dim these attributes in the code to avoid becoming distracted by them.

To begin work on the `length` function, Baklava now creates an example call to the function: on the canvas, she double-clicks to add new code and types `length int_list`. As before, a new binding is inserted in the code (named `length_int`) and an associated tangible value (TV) appears on the canvas (Figure 3, right).

The `length_int` TV has two differences from the previous `int_list` TV. First, its result value (displayed as `?`, explained below) has a yellow background – this indicates the result is *not* simply a constant introduced in the expression: it came from computation elsewhere. Second, the `int_list` variable usage in the TV's expression bears a superscript indicating the value of `int_list`, namely `[0; 0; 0]`.

In Maniposynth, using an undefined variable – in this case, `length` – automatically inserts a new let-binding (TV) for that variable. Because Baklava used `length` as a function, a new function skeleton was inserted in the code (`let length x1 = (??)`).

Function TVs are displayed specially on the canvas (Figure 4). Immediately below the function name, a *function IO grid* displays the function input and output values encountered during execution. Immediately below the IO grid is a blank white area which is a *subcanvas*

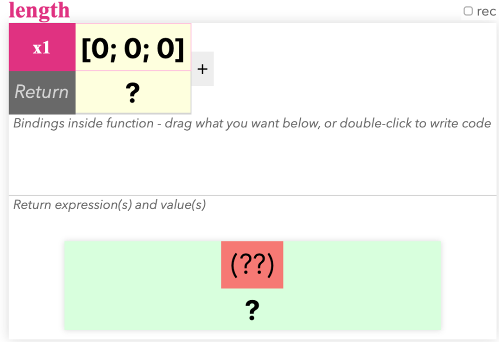


Figure 4 Tangible value for the function skeleton binding `let length x1 = (??)`.

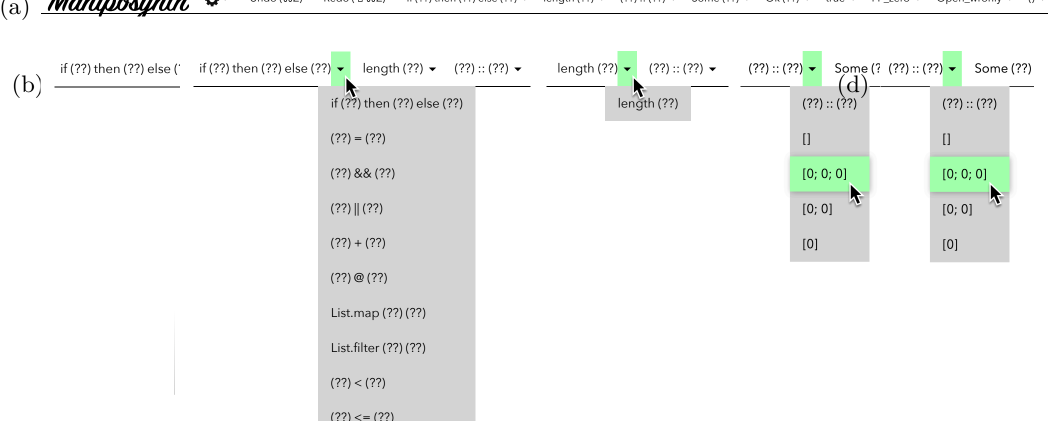


Figure 5 (a) The toolbar, including menus for (b) skeleton expressions, (c) functions defined in the current file, and expressions auto-generated from data types in scope, e.g. the lists shown in (d).

for the bindings (TVs) inside the function, of which there are none yet. Below the subcanvas is a (non-movable) TV for the return expression and overall result value of the function. Currently the function return expression is a *hole expression*, written `(??)`. Hole expressions are placeholders, expected to be filled in later. For this reason, they are displayed larger than normal expressions (to make them easier targets for clicking) and have a slowly pulsing red background (to remind the user that the program is unfinished). While the `(??)` syntax is supported by OCaml's editor tooling (Merlin [6] and its language server protocol wrapper [26]), programs with holes are ordinarily not executable. To continue to provide live feedback in the presence of holes, MANIPOSYNTH evaluates hole expressions `(??)` to a *hole value*, displayed as `?`. This hole value `?` is the current return value of the function shown below `(??)` – in green because it was introduced by the immediate expression above – and also shown in the "Return" row of the IO grid as well as, back on the main top-level canvas, in the result value of the `length int_list` function call.

Baklava does not like the default `x1` parameter name in the `length` function and wants to rename it. Most items in MANIPOSYNTH can be double-clicked to perform a text edit. Baklava double-clicks the pink-background `x1` to rename the variable (patterns are pink), and writes the name `list` instead. Figure 6a shows the code at this point.

Figure 6 Creating a recursive call. (a) Code before. (b) Dragging a new `length` call into the `length` function. (c) Resulting code and (d) resulting `length` function TV.

A goal of Maniposynth is to allow non-linear editing – the ability to make incremental progress toward a solution. Baklava knows she must make a recursive call to the `length` function, so, without thinking hard about what might come after, she decides to add `length (??)` inside `length`. She could double-click and type this code, but typing `(??)` requires some finger gymnastics. Maniposynth supports a large number of drag-and-drop interactions. Any green expression can be dragged to a new location to duplicate that expression: dropping on an existing expression (e.g. a hole) replaces the existing expression, while dropping on a (sub)canvas inserts a new binding (TV). Values and patterns can also be dragged to expressions or (sub)canvases – when hovering over a value or pattern, a tooltip shows what expression will be inserted. Finally, a *toolbar* at the top of the window (Figure 5a) offers menus containing skeleton expressions: the first menu offers common expressions such as `if (??) then (??) else (??)` (Figure 5b); the second menu offers functions defined in this file (Figure 5c); and the remaining menus offer constructors and automatically generated example values of the types in scope (e.g. Figure 5d; the expressions are the same as those offered by autocomplete). User-defined custom data types, if any, also appear as menus.

Baklava drags `length (??)` from the toolbar into the subcanvas for her `length` function (Figure 6b). A `length2 = length (??)` binding is created in the code (Figure 6c) and an associated TV appears inside `length` (Figure 6d). Maniposynth also changes the top-level `let length = ...` into `let rec length = ...`.

Because `(??)` produces hole value `?` instead of crashing, the `length` function is now diverging as `length (??)` calls `length (??)` which calls `length (??)` and so on. Maniposynth uses fueled execution to cut off infinite loops and keep functioning. In the function IO grid, extra columns show these calls (Figure 6d), but other than understanding why these extra columns are there, Baklava need not mind that her program is momentarily divergent.

(a)

(b)

```
int list
let int_list = [ 0; 0; 0 ] [@@pos 268, 93]

'a list -> 'b
let rec length list =
  let length2 = length (??) [@@pos 55, 12] in
  match list with hd :: tail → (??) | [] → (??)
  [@@pos 85, 233]

'a
let length_int = length int_list [@@pos 481, 91]
```

(c)

```
'a list -> 'b
let rec length list =
  match list with
  | hd :: tail →
      let length2 = length tail [@@pos 73, 13] in
      (??)
  | [] → (??)
```

Figure 7 (a) A “Destruct” button appears when the cursor hovers over an input that is an ADT value. (b) Code and TV for `length` function after destructing the `[0; 0; 0]` input value. (c) After dragging the tail value `[0; 0]` to the red hole argument `(??)` for the recursive `length` call.

Baklava wants the recursive call to operate on the tail of the input list. When she moves the cursor over the input list in the IO grid, a “Destruct” button appears (Figure 7a), which she clicks. As shown in Figure 7b, a `match` statement (i.e. case split) is added, with holes for the return expression of each branch. On the display, there are a number of visual changes. In the IO grid, `hd` and `tail` pink subscripts appear inside the input list `[0; 0; 0]`, labeling the subvalues that are now bound to names by the `match` statement. To make these bindings even clearer, they are also represented as two new TVs in the function subcanvas. Finally, the function now has two possible return expressions: both appear as (non-movable) TVs at the bottom of the function, one is grayed out indicating it is not the branch taken when the input is `[0; 0; 0]` (the column currently selected in the IO grid). Above the two return TVs is an indication of the scrutinee, “← list →”, which allows editing of the scrutinee expression.

Now that the list tail is exposed on the subcanvas, Baklava drags it (either the pink `tail` name or the `[0; 0]` value below it) onto the hole in `length (??)`, transforming it into `length tail`. In her code, the binding is moved from the top-level of the function into the branch in which `tail` exists (Figure 7c). Because Maniposynth embraces non-linear editing, the user should not have to worry about binding order – bindings will automatically be shuffled around as necessary to place items in the appropriate scope.

The additional calls from the recursion appear in the function IO grid, each still returning hole value `?` (Figure 7c, right). Baklava would like to edit the base case, so she looks for the column in the IO grid where the input is `[]`, and then clicks that column to bring that *call frame* into focus. Call frames are effectively equivalent to runtime stack frames. The TVs not executed on that call are grayed out (`hd`, `tail`, `length2`, and the return for the `hd::tail`

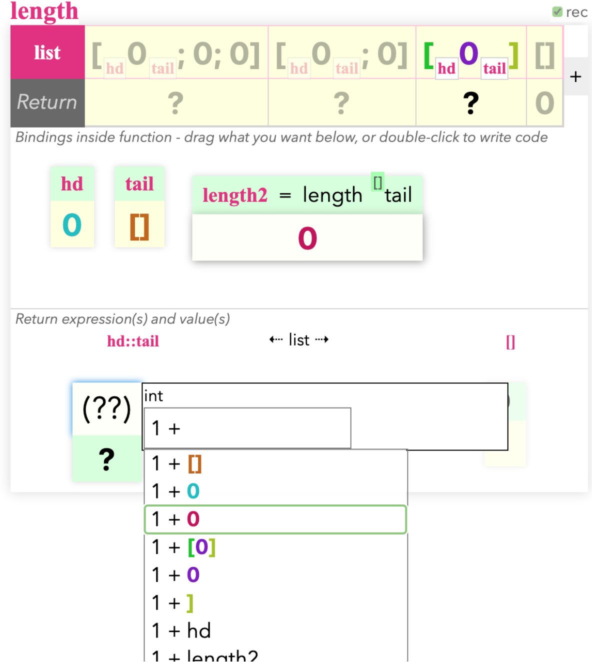


Figure 8 Autocompleting to a value in scope.

branch). Baklava double-clicks the no longer grayed-out return expression `(??)` for the base case and sets it to the constant `0`. (She could also have double-clicked the green-background hole value `?`; values are rendered with a green background when double-clicking them will effect an edit on the expression immediately above.)

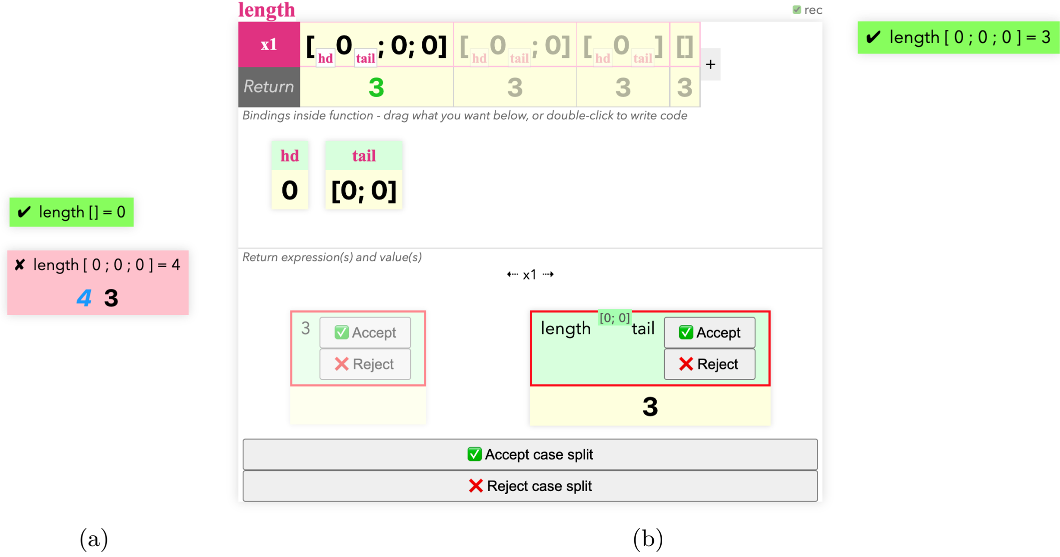
Baklava now clicks the second-to-last call frame in the IO grid to bring into focus the call where the input is `[0]`. The return expression for this branch is still `(??)`. She notes that the TV for the `length tail` call now displays a result value of 0. Baklava double-clicks the return expression `(??)` and, after typing "`1 + `" she pauses (Figure 8). When she began to type, Maniposynth recolored the displayed values in scope using different colors, and now, looking at the autocomplete options, she sees `1 + 0`, `1 + 0`, and `1 + 0` among the possible autocompletions – each with a different color `0` corresponding to a similarly colored `0` value elsewhere on screen. The maroon `0` is the return from `length tail`, so she chooses that. The branch return expression becomes `1 + length2`, and Baklava can now see in the IO grid that her function returns the correct value for all inputs (Figure 1).

2.2 Undo and Delete

Maniposynth supports undo/redo. Additionally, any expression may be selected by a single click and then transformed to a hole by pressing the Delete key. Entire let-binding TVs can similarly be selected and deleted, removing them from the program. Uses of the binding must be deleted before deleting the binding itself – otherwise Maniposynth will immediately recreate a binding to satisfy the unbound variable uses.

2.3 Value-Centric Shortcuts, and Synthesis

There are usually multiple ways to complete a task in Maniposynth. Below are a few variations Baklava might have performed instead.



(a) (b)

Figure 9 (a) A satisfied and unsatisfied assertion. (b) An undesired synthesis result. After rejecting the return expressions (or the entire case split), the next result will be correct.

Drag-to-Extract. To extract the list tail for use in a recursive call to `length`, Baklava clicked "Destruct" on the input value and dragged the resulting `tail` name to the `length (??)` call. The explicit "Destruct" step can be skipped. Because MANIPOSYNTH aims is to make values live as much as possible, *subvalues* can also be manipulated. Baklava could have hovered over the portion of the input list `[0; 0; 0]` that is the tail of that list, namely `; 0; 0]`, and dragged that subvalue directly to her `length (??)` call without pressing "Destruct". The destruction will be performed automatically, producing the same code.

Autocomplete-to-Extract. Similarly, visible subvalues are also offered as autocompletions. Perhaps the fastest way to create list `length` is, immediately after the `length` function skeleton is created, to double-click the return hole expression, type "`1 + length `", and then finish the new expression by selecting `; 0; 0]`, the tail of the input list, from the autocomplete options. The expression `1 + length tail` and the needed pattern match will be inserted, leaving only the base case to fill in.

Assertions. Baklava started with an example call to `length`. To remind herself of the goal, she could have created an assertion instead: typing `length [0; 0; 0] = 3` on the top-level canvas will add an `assert` statement instead of a named binding. Assertions are rendered in red when unsatisfied, and both the expected result (in blue) and the actual result (in black) are shown. When satisfied, an assertion turns green and its result is hidden (Figure 9a). Assertions can also be added via the function IO grid: clicking the "+" button at the right of the IO grid creates a new column in the grid, allowing Baklava to fill in input values and expected output. Upon hitting enter, the column is reified by adding a new assertion at the top-level, so that the function is indeed called with the specified arguments.

Program Synthesis. Assertions facilitate *programming by example (PBE)* [28], a workflow currently available in Microsoft Excel [27] but not yet in ordinary programming settings. After asserting `length [0; 0; 0] = 3`, Baklava might have clicked the "Synth" button in

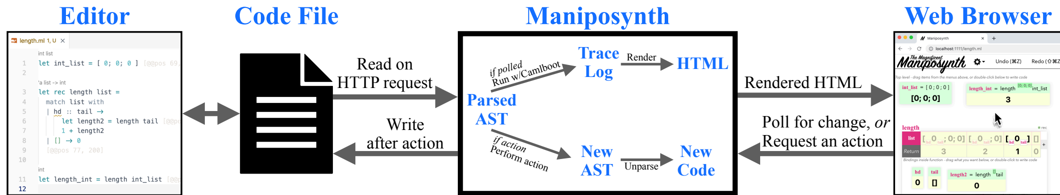


Figure 10 MANIPOSYNTH architecture overview.

the lower-right corner of the UI. MANIPOSYNTH will use type-and-example based synthesis (inspired by MYTH [59]) to guess hole fillings until the assertion is satisfied or the synthesizer gives up (after between 10 and 40 seconds). The synthesizer incorporates a simple statistics model and other heuristics to improve result quality (Section 3.4). In this scenario, with only the single assertion, MANIPOSYNTH instantly finds a filling that creates the proper case split, but places `3` in the base case and `length tail` in the recursive case (Figure 9b). The result is incorporated into the code, but presented to Baklava with buttons prompting her to "Accept" or "Reject" parts of the synthesized expression. Rejected expressions are transformed back into holes, with an annotation telling the synthesizer to avoid that expression in the future. Baklava accepts the case split, but rejects its two return expressions. When she clicks "Synth" again, the desired return expressions are discovered she accepts them.

3 Implementation

The main features of MANIPOSYNTH now demonstrated, next we describe how it works.

3.1 Architecture Overview

MANIPOSYNTH is a web application written in ~8600 lines of OCaml (excluding the interpreter) and ~2000 lines of Javascript. OCaml's compiler tools and AST data types are used to handle parsing, type-checking, type environment inspection, and pretty printing of modified code. Modified code is further beautified by running it through `ocamlformat` [4] if the user has it installed. Comments are (unfortunately) discarded by OCaml's parser.

For displaying live feedback, we need to run the program and log the runtime values flowing through the code. We modified the OCaml interpreter from the Camlboot [12] project to emit a trace of all runtime values at all execution steps. We also performed additional modifications to handle holes and assertions (described in the next section).

After MANIPOSYNTH runs the code via our modified Camlboot, MANIPOSYNTH associates runtime values from the logged execution trace with program expressions, and then renders HTML which is displayed in the browser. Almost all OCaml-specific logic is handled server-side. The client-side Javascript only handles TV positioning and standard GUI interaction logic. When the user performs an action, the Javascript sends the action to the server via HTTP, the code is modified on disk, and the server prompts the browser to reload the page to re-render the display. The browser also polls the server via HTTP so that when the file is changed on disk, the display will refresh. This overall architecture is outlined in Figure 10.

Below, we describe our modified Camlboot interpreter, then how bindings are reordered to provide a non-linear experience, and lastly the mechanics of the synthesizer.

3.2 Interpreter

MANIPOSYNTH needs to provide live runtime values. We base MANIPOSYNTH on the OCaml interpreter in the Camlboot [12] project, an experiment in bootstrapping the OCaml compiler. The Camlboot OCaml interpreter is written in OCaml and represents all values as an ordinary OCaml algebraic data type (ADT), which allows inspecting their type and structure at runtime, at the cost of somewhat slower execution relative to compiled code. We modified Camlboot to handle holes and assertions, and to log runtime values during execution.

Supported Subset. Unmodified, the Camlboot interpreter will run a large subset of OCaml. The tooling and display in MANIPOSYNTH, however, currently only fully supports a smaller subset, shown in Figure 11. At the top-level, programs in MANIPOSYNTH are expected to consist only of type declarations followed by (potentially recursive) let-bindings; assertions are expected to occur only at the top-level. Only single-name patterns have full UI support (although internal operations such as free variable analysis account for nested patterns). Supported expressions include holes, base value constants, argument-less, single-argument, and multi-argument constructors, variable usages, function introductions with an unlabeled parameter, multi-argument function applications, (potentially recursive) let-bindings, tuples, if-then-else, and pattern match case splits. Case splits are only fully supported on constructors.

Records do not have complete UI support. User-defined modules, opening modules, imperative functions, and object-oriented features are currently unsupported.

The swath of supported syntax was enough to cover the kinds of data structure manipulations we explored in our evaluation. During the user study exercises, participants rarely missed the unsupported syntax. Even so, for the tool to become practical for everyday use, the users noted it would need to support modules and imperative programming.

Holes and Bombs. It is best for the user if live feedback is available even if the program is incomplete. While we could have the interpreter crash on the first hole, that may still be too restrictive, e.g. if the expression is new and is still dead code, then the presence of the hole should be inconsequential to the rest of execution. A thorough solution would be to adopt the Hazelnut Live semantics, which describes how to evaluate *around* holes [58]. When holes reach elimination position, terms become stuck (e.g. What should hole plus hole be? Or which case branch should we take when the scrutinee is a hole?). Hazelnut Live evaluates around the term by, effectively, turning the stuck term into a value which is propagated until it causes another term to become stuck, and so on. While this can offer intriguing UI possibilities in its own right (outlined in [58]), it requires displaying stuck terms to users as if they are values. MANIPOSYNTH may do so eventually, but our display is already full of elements to keep track of. Asking users to make sense of stuck terms, displayed far from their origin, might be confusing.

$$
\begin{array}{rrcl}
\textbf{Programs} & P & ::= & \overline{\textbf{type}\ t\ =\ T}\ \ \overline{B} \\
\textbf{Types} & T & ::= & \text{(from OCaml)} \\
\textbf{Top-Level} & B & ::= & \textbf{let}\ x = e \\
\textbf{Bindings} & & | & \textbf{let rec}\ x_1 = e_1 \\
 & & | & \textbf{let}\ () = \textbf{assert}\ (e_1 = e_2) \\
\textbf{Patterns} & p & ::= & C \mid C\ x \mid C\ (x_1, \overline{x_i})
\end{array}
$$

$$
\begin{array}{rrcl}
\textbf{Exp.} & e & ::= & (??) \mid c \mid C \mid C\ e \mid C\ (e_1, \overline{e_i}) \\
 & & | & x \mid \textbf{fun}\ x \rightarrow e \mid e_1\ \overline{e_i} \mid x\ \overline{e_i} \\
 & & | & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
 & & | & \textbf{let rec}\ x_1 = e_1\ \textbf{in}\ e_b \\
 & & | & (e_1, e_2, \overline{e_i}) \\
 & & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
 & & | & \textbf{match}\ e_1\ \textbf{with}\ \overline{p \rightarrow\ e_i}
\end{array}
$$

Figure 11 The subset of OCaml fully supported by MANIPOSYNTH. Overlines denote zero or more instances of the syntactic element. Unsupported expressions and patterns are displayed but do not have full UI support. The synthesizer (Section 3.4) emits only those forms displayed in blue.

MANIPOSYNTH adopts a middle ground that does not allow terms to become stuck. The hole expression `(??)` introduces a hole value `?` that remembers the introduction location and captures a closure. (This closure is not displayed, but is occasionally used during synthesis.) Hole values propagate through evaluation. If a hole value reaches elimination position (e.g. `? + ?`), we resolve the expression to a special Bomb value (displayed as 💣). Similarly, if a Bomb reaches elimination position, another Bomb is produced. In this way, execution continues and expressions unrelated to the unfinished code still provide live feedback.

Finally, to prevent infinite loops from inhibiting live feedback, MANIPOSYNTH uses fueled execution to abort when the right-hand side of a binding takes too long to execute. Each top-level let-binding is allocated 1000 units of fuel (execution steps), and each non-top-level let-binding reserves 50 units for later execution in case the binding diverges. When the interpreter runs out of fuel on a binding, all patterns at the binding are bound to Bomb, and execution continues if any fuel remains. Divergence is moderately common, because recursive call skeletons like `length (??)` from the Overview Example will repeatedly call the function with a hole value. Thus it is important that execution bypasses the divergence with some fuel reserved so that later TVs will still show live values in the display.

Assertion Logging. When an assertion is encountered during execution, ordinarily OCaml would throw an exception if the asserted expression returns false. Instead, MANIPOSYNTH logs the result for later, but never raises an exception. Only equality comparisons are supported for now; unsupported assertions are skipped. The expected expression (the right-hand side), the subject expression (the left-hand side), and the result values of both are logged. Logged assertions are used both for synthesis and to display blue expected values to the user wherever that same expression and value is encountered during execution (Figure 9a).

Tracing. In addition to assertions, MANIPOSYNTH also logs other execution information needed for display. Our interpreter records information in two places: each execution step is entered into a global log, and we tag side information onto runtime values.

At each execution step and at each pattern bind we add a log entry to a global trace, recording the current AST location, the call frame number (from a global counter incremented upon each function call), the result value or value being pattern matched against, and the execution environment of bound variables. When producing the display, this information is queried to discover which values flowed through which locations and under which call frames, and the appropriate values are rendered.

For convenience, we also store extra information on values. On values we log the type of the value when it is introduced (or returned from a built-in external primitive such as addition) so we have a concrete type associated with the value even if it is later used in a polymorphic context. To the value we also attach a list of frame numbers and AST locations of the expressions and patterns the value passes through, to, e.g., conveniently interrogate where a value was first introduced. For example, to display function closure values, we find where the closure was bound to a name and display that name as the rendered value.

These mechanisms are sufficient to render the live feedback in MANIPOSYNTH. Extensive logging might be expected to slow down execution. But when applied to the small programs tested at present, HTML rendering tends to take longer than the initial execution. Overall, the MANIPOSYNTH server is generally able to provide a response in under 200ms.

```
let a = 1

let c () =
  let x = (a, b, c, d) in
  let a = 0 in
  x

let b = 2
```

(a)

```
let a = 1

let c () =
  let x = (a, b, c, d) in
  let a = 0 in
  x

let b = 2
```

(b)

```
let a = 1

let b = 2

let rec c () =
  let a = 0 in
  let d = (??) in
  let x = (a, b, c, d) in
  x
```

(c)

Figure 12 (a) A program. (b) How MANIPOSYNTH interprets what each variable usage refers to. (c) The reordering MANIPOSYNTH performs. The bindings of `a` and `b` are moved up so they are in scope, `c` is marked as recursive, and `d` is created.

3.3 Fluid Binding Order

A primary goal of MANIPOSYNTH is to offer a non-linear editing experience. We do not want users to have to worry about binding ordering. If the user sees a name on the 2D canvas, they should be able to reference that name in the expression they are editing even if, in the written code, that name is introduced later in the program.

To support this non-linear workflow, only limited variable shadowing is supported. All names are assumed to be unique within each top-level definition. Names may (initially) be out of order. Figure 12a presents an example. As shown in Figure 12b, MANIPOSYNTH interprets each of the variable uses in `(a, b, c, d)` different from standard OCaml. Although there is a top-level binding `a = 1` in scope, the use of `a` in `(a, b, c, d)` refers to the binding `a = 0` below it, because MANIPOSYNTH is agnostic about the order of names within a top-level definition (within the top-level definition `c`). The variable `b`, however, does not have a definition within `c`, so its usage refers to the top-level binding `b = 2`, despite being (momentarily) out of scope. Similarly, the usage of `c` refers to the top-level binding for `c`, although the binding is not (yet) marked recursive. The variable `d` has no binding.

To reify these references as valid OCaml, after every user action MANIPOSYNTH reorders bindings, move bindings into `match` statement branches (not shown here), and adds a `rec` flag on bindings that refer to themselves. Only single recursion can be inferred for now (multiple recursion may be added manually in the text editor). Figure 12c shows the result for this code example. The nested definition of `a` is moved before `x`, the top-level definition of `b` is moved up as well, the binding for `c` is marked as recursive, and a new definition is added for the missing variable `d`.

When creating a new definition for an undefined variable, if it is used as a function in an application the new variable is bound to a function skeleton with the appropriate number of parameters. Here, `d` is not used as a function and is initialized as a simple hole `(??)`.

The overall effect of the above reordering is that users rarely need to think about binding order in their code. They can use the displayed TVs as if they are all visible to each other.

3.4 Synthesizer

As discussed in the Overview Example, MANIPOSYNTH includes a programming by example (PBE) workflow to help users finish their incomplete code. Here we detail the program synthesizer's operation and our design choices in its implementation.

The Maniposynth synthesizer does not contain any new "big" ideas, but the design was carefully chosen for our setting. To be as practical as possible, we had four goals:

(a) Few examples. To reduce user burden, we would like the synthesizer to operate with few examples. For example, Myth [59] also targeted an OCaml subset, but required that user-provided examples include all needed recursive calls – e.g. `length [0;0] = 2`, `length [0] = 1`, and `length [] = 0`. This "trace completeness" requirement is burdensome; we would like our synthesizer to operate with only one or two examples.

(b) No type annotations. Similarly, Myth and its successor Smyth [48] require holes to have types before synthesis, which requires manual annotation. We would like to relieve the user of this responsibility and operate without explicit type annotations.

(c) As simple as possible. The primary goal of Maniposynth is to explore non-linear editing, not synthesis per se, so we wanted to keep our synthesizer as simple as possible. For now, we did not adapt Smyth because, although it appropriately relaxes the trace-completeness requirement, Smyth utilizes a complicated synthesis schedule and requires the Hazelnut Live machinery [58] for evaluating around holes.

(d) Quality results. When given only a few examples, synthesizers are notorious for producing simple but undesirable results (for example, "January, Febuary, Maruary" [2]) which limits their utility. This problem is compounded when the synthesizer is asked to operate in practical environments with many variables in scope, rather than unrealistic bare minimal execution environments often used for synthesizer benchmarks. Our synthesizer should operate with the OCaml standard `Pervasives` library open in the execution environment so the synthesizer may choose to use, for example, addition and subtraction. We adopt statistics and heuristics to make this tractable.

Myth used types and examples to dramatically reduce the search space and to intelligently introduce case splits. To meet the above goals, we built a Myth-like synthesizer, but we relax the trace-completeness requirement and instead rely on a statistics model to guess more likely terms sooner. Our target language subset, the statistics model, the synthesis operation, and our other heuristic choices are described below.

Synthesizable Subset. The blue expression and pattern forms in Figure 11 describe the OCaml subset that the synthesizer may produce to fill holes in the program. It can introduce functions, `match` expressions, constants (drawn from a corpus), variable uses, function calls with a variable in function position, constructor uses, and if-then-else expressions.

Statistics Model. Naively, guess-and-check will produce a large number of programs the user is unlikely to want. Incorporating a statistics model guides the synthesizer to guess more likely programs sooner and can speed up synthesis by multiple orders of magnitude [45, 39]. It also has the potential to offer more reasonable results when fewer examples are given.

We model program likelihood using a probabilistic context-free grammar (PCFG). A PCFG is a grammar with a probability assigned to each production rule. For our synthesizer, we derived the probabilities of the production rules from a corpus of OCaml code – namely, the source files required to build the OCaml native compiler. The overall probability of a program term is the product of the probability of the production rule of the term with (recursively) the probability of all its subterms. Identifiers are handled specially: an identifier's probability is based on how spatially close it is to its introduction pattern in the code.

For example, the most likely term (i.e. what the synthesizer should guess first) is always the most recently introduced variable. The production rule for an identifier has a probability of 52%, the probability that the identifier is local is 73%, and the probability a local identifier is the most recently introduced variable is 31%, for an overall probability of 12%.

Type-Directed Refinement. Myth divides synthesis into two processes. *Type-directed refinement* introduces program sketches – either data constructor applications, function introductions, or case splits – at holes based on the type at the hole and the types of variables in scope (to find an appropriate scrutinee for a `match`). These sketches contain further holes to fill (i.e. for the function body and the match branches). Type-directed refinement alternates with *type-directed guessing*, which performs simple type-constrained term enumeration to fill remaining holes (guessing will not introduce functions or `match` statements).

Maniposynth uses type-based refinement to introduce functions and insert case splits (data constructors are instead handled in the guessing process). Maniposynth refines a hole into a function zero to three times (i.e. supporting up to three arguments) before possibly introducing a single case split. The user can add further case splits with the "Destruct" button. Introducing functions is rarely needed in practice because, in the Maniposynth UI, undefined variables are inserted with a function skeleton.

Type-Directed Guessing. After refinement, terms are enumerated (guessed) at holes up to a probability bound [44] according to the PCFG. During term enumeration, the probability bound is treated as a resource. When the probability is exhausted, no further enumeration occurs on a subtree. If a candidate subterm's probability is above the final bound, the remaining probability is available for enumerating sibling terms.

Within a hole, term enumeration is type-directed, starting from the type of the hole. Leveraging OCaml's type checking machinery, subterms are unified during the enumeration process to narrow the type. For example, if a hole has type `int` and the synthesizer guesses a call to `max`, of type `'a → 'a → 'a`, the return type will be unified with `int` and the synthesizer will only guess terms of type `int` for the arguments.

Initial sketches often have polymorphic types unhelpful for synthesis. For example, the `length` function has type `'a → 'b` in `let length x1 = (??)`. To tighten these bounds before term enumeration, input and output types of functions are speculatively chosen based on the examples. If the user asserts that `length [0] = 1`, then `length` is given the speculative type `int list → int`. Speculative types are removed after term enumeration in case the final code has a more general inferred type (`'a list → int` in this case).

To produce more natural results, Maniposynth limits where constants may appear. A term is estimated to be non-constant if it uses an introduced function parameter or variable introduced under the outermost enclosing function. At most one hole may be constant, and, when introducing a function call, at least one argument must be non-constant. Term enumeration avoids constants in locations where inserting one would violate these rules.

Non-Linearity. To follow the non-linear behavior of Maniposynth, the guessing process also guesses variable names that are not in lexical scope but could be moved into scope via Maniposynth's binding reordering algorithm (Section 3.3). The reordering algorithm is applied before testing whether a candidate program satisfies all assertions.

Final Heuristics. Finally, when all holes have been filled with type-appropriate terms within the probability bound, the candidate program is accepted if:

(a) All assertions are satisfied. (Fueled execution is used when checking assertions.)
(b) At most one hole is filled with a constant.
(c) All introduced function parameters are used.
(d) The result at a hole has not previously been rejected by the user.
(e) Execution of the examples encounters all filled hole locations (i.e. the execution path does not somehow avoid a hole).

If no satisfying hole fillings are found at the initial probability bound and a 10 second timeout has not been reached, guessing is restarted with a new bound 1/20 of the old. If there is a valid candidate program, the program with highest probability is returned. Enumeration within a given probability bound is not precisely from highest to lowest probability, however, so a timeout will not interrupt a round of synthesis until the full space of that probability bound is explored. Thus, the timeout the user experiences varies between 10 and 40 seconds.

4 Evaluation

To evaluate to what degree Maniposynth meets its goal of providing value-centric, non-linear editing, we performed two evaluations. In one, an expert user (the first author) used Maniposynth to implement 38 functions from the exercises and homework of a functional data structures course [56]. In the second, to provide additional qualitative insights on the operation of the tool, we hired two professional OCaml programmers, guiding and observing them as they used Maniposynth to implement a subset of the above functions.

4.1 Study Setups

The first six lessons of the course [56] cover natural numbers (via an ADT), various list functions, leaf trees, binary trees, binary search trees, and a form of binary search tree that also records on each node the minimum value of all its descendants. We excluded the six functions on this specialized tree because of time constraints. The course exercises and homework spanned 38 functions on the remaining data structures. The first author implemented each of these functions in Maniposynth with the code editor hidden. Maniposynth was configured to log the number and kinds of actions performed. We report on these in the next section. Our aim was to show that the Maniposynth interface was able to implement these exercises and to discover if there were any obvious trouble points.

For our user study, we advertised on `https://discuss.ocaml.org/` and hired two professional OCaml programmers for three sessions each. Sessions were spread over three weeks, with each session lasting two hours. Participant 1 (P1) and Participant 2 (P2) had 5 and 11 years, respectively, of professional OCaml experience. The participants ran Maniposynth on their own computers alongside their preferred text editor (Vim for both). The study facilitator connected via video conference and recorded the sessions. Participants implemented their choice of exercises from the list, or suggested their own task to complete. The facilitator provided varying amounts of guidance throughout, starting with close guidance to teach the tool and transitioning to less intervention as participants became more comfortable. After each exercise and at the end of each session, participants discussed a series of questions posed by the facilitator. In concert with Maniposynth's four design principles – value-centric operation, non-linearity, supporting synthesis, and bimodality – we aimed to gain insights about the following four research questions, along with three supplemental questions:

RQ1. How do users interact with the live values?
RQ2. How do users work non-linearly?
RQ3. How do users interact with program synthesis?
RQ4. How do users interact with their text editor?
SQ1. What are the pain points? How might the system be improved?
SQ2. Are participants comfortable enough to complete a task without guidance?
SQ3. What else can we learn via the lenses of the Cognitive Dimensions of Notations [25]?

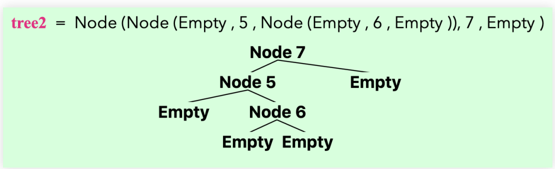


Figure 13 Maniposynth beautifies tree-like values.

For each participant, the first session introduced the tool without synthesis, the second session introduced synthesis (before the synthesizer had a statistics model), and the third session concluded with the tool as presented here. Based on participant feedback, we fixed bugs and made improvements between each session.

4.2 Results

Example Corpus Implementation. The expert implementer spent about 4.5 hours total – working in fullscreen with their text editor hidden – implementing the 38 functions, resulting in about 550 lines of code (including AST annotations and examples written for live feedback, but excluding whitespace). A quantitative summary of these example exercises is shown in Table 1. There are often many paths to a correct implementation, so to constrain the workflow the implementer did not use synthesis, and did not use the ordinary text editor except to copy an earlier function into a later exercise (in case of dependencies) or when Maniposynth crashed on the given code (e.g. when the implementer tried to raise an exception in unreachable code that turned out to be reachable!). Note that for the functions operating on tree-like data types (ADTs with multiple children of the same recursive type), Maniposynth's live display helpfully draws the trees as trees (Figure 13).

Primarily, these 38 examples show that the Maniposynth UI is expressive enough to create these programs. We also noticed two qualitative takeaways from the exercises. First, although we believe bimodality is an important property for the grounding and long-term practicality of the tool, it is possible to hide the text editor and work entirely in Maniposynth. Second, even with live feedback available, it is not always used – a theme that reemerged in our user study. We now discuss these two observations in more detail.

The non-linearity machinery largely worked – the implementer did not have trouble with binding order. Even so, they were careful to name extracted subvalues well, because the positioning of the extracted TVs on the 2D display did not (by default) reflect the items' positions in the original data structure. Particularly for nested matches, there were sometimes a large number of these extracted values displayed and it was hard to keep track of them. The implementer found it helpful to reposition the extracted TVs (the TVs representing case split branch patterns) to reflect those original positions. Ordinary textual code for case split patterns would provide some of these positional cues without manual interaction.

On a few exercises requiring nested `match` statements, Maniposynth initially created the wrong nested `match` structure; with the non-linear display, this is hard to notice and requires thinking about the `match` nesting structure shown in the return TVs area. Regardless, the implementer worked around the trouble by undoing and triggering the destructions differently.

The second observation from these examples is that the implementer noticed that they seem to flip between two mental modes: these modes correspond roughly to focusing on displayed values versus focusing on expressions. In the *value-oriented* mode, the implementer would put their attention on the live values to consider if the code is operating correctly;

Table 1 Example exercises, with lines of code, number of assertions, time in minutes, number of mouse actions (excluding selection and undo/redo), number of keyboard interactions (e.g. typing in a textbox), number of undo/redo/deletions, number of type errors encountered, and number of times MANIPOSYNTH crashed and the file had to be repaired in the text editor.

Function	LOC	Asserts	Time	Mouse	Keybd	Un/Re/Del	TypeErr	Crash
nat_plus	5		0.8	6	5			
nat_minus	8		1.9	6	11			
nat_mult	9		1.4	8	6			
nat_exp	13		2.1	9	6			
nat_factorial	13		1.6	8	4			
nat_map_sumi	10		2.6	11	5		1	
count	9		1.9	9	11			
length	4		0.3	1	7			
snoc	8	1	2.4	8	12	2		
reverse	8		1.5	4	9			
nat_list_max	17		4.6	23	21			
nat_list_sum	13		1.1	9	4			
fold	9		3.2	14	6			
shuffles	14		14.5	25	28	2		
contains	9		2.2	10	13	1		
distinct	16		2.4	9	11	2		
foldl	10	1	1.5	10	6		1	
foldr	8	1	1.8	10	5			
slice	12	3	9.8	19	22	4		
append	8	1	1.4	7	9			
sort_by	21	3	6.2	17	29			
quickselect	13	1	13.1	19	38	1	1	
sort	16	3	5.6	11	32	2		
ltree_inorder	12	1	2.9	7	20	1	1	
ltree_fold	13	1	3.1	13	13			
ltree_mirror	11	1	4.4	12	6		1	1
bst_contains	14	3	6.6	11	32	1		
bst_contains2	17	5	10.4	20	41	2		
btree_join	34	2	61.7	82	64	51		2
bst_delete	36	2	14.4	31	24	4		
bstd_valid	29	3	32.2	63	100	4	1	
bstd_insert	18	2	8.0	38	23	3		
bstd_count	21	1	7.6	15	32	1		
bst_in_range	31	3	9.3	23	39	3		
btree_enum	29	3	19.2	31	51	6	3	
btree_height	15	1	1.9	11	14			
btree_pretty	14	1	3.7	4	21		4	
btree_same_shape	19	1	8.1	14	34	7		
Total	**566**	**44**	**277.6**	**628**	**814**	**97**	**13**	**3**

in the *expression-oriented* mode, the implementer would read expressions and simulate the computer's operation in their head. As a matter of discipline, the implementer was trying to push themselves to consider and use the live values, but nevertheless often reverted to thinking only about the expressions instead. We have three hypotheses for why there seems to be a tendency to revert to focusing on expressions instead of values.

Hypothesis A: Expressions are a concise language to represent abstractions, and programming is, fundamentally, abstract. Concrete values do not directly represent the abstraction.

Hypothesis B: Seasoned programmers have many years of experience reading code and simulating the computer in their head. Our brains have adapted to it, and it feels natural.

Hypothesis C: MANIPOSYNTH did not provide enough live feedback and forced the implementer to consider the expressions. In some cases this was immediately true: MANIPOSYNTH currently only displays the first and last three call frames, with no option to see the others, so sometimes relevant values were in unavailable call frames. Additionally, when initially trying to figure out what algorithm was needed at all, the implementer found it easier to work out the initial algorithm sketch in their head rather than guess and check in MANIPOSYNTH.

All three reasons likely contributed to the tendency to put attention back on expressions rather than values. A similar theme was observed in the user study, which we now discuss.

RQ1. How do users interact with the live values? Value-oriented focus versus the "old way" of expression-oriented focus is a theme that appeared in several participant interactions. For example, despite values featuring prominently in the display, it took until after the entire first exercise for P2 to fully realize they were looking at and working with live *values*. In another scenario, P1 and the facilitator together spent an embarrassingly long time trying to find a bug in an `insert_into_sorted_list` helper. After finding the bug they realized that, had they carefully inspected the live values, they might have found the bug much sooner. Additionally, P2 observed that they are so used to reading trees as long lines of text (e.g. `Node (Leaf 2, Node (Leaf 2, Leaf 3))`) that they were subtly repelled by the "much more readable" beautified 2D rendering of tree values (Figure 13).

Even so, participants still did use the live display and expressed appreciation for it. For example, P2 noted that the live display ameliorated the need to write large amounts of tree pretty-printing code to perform printf-debugging.

Live values require the user to switch call frames to see other example function calls, or calls that hit a different branch in the code. This was not always natural for participants. In the first session, P2 admitted to sometimes being confused about what branch they were looking at. And, despite gaining moderate proficiency with the tool by the end of the study, P2 still remarked that it was hard to think about how you can flip between frames. How to modify the display to help clarify this operation remains an open question.

RQ2. How do users work non-linearly? We want to know how programmers adapt to MANIPOSYNTH's non-linear style. The tool requires a number of "inside-out" (P1) changes in thought, such as creating an example before defining a function, providing expressions *without* naming them first, and not worrying about let-binding order. By the end of the study, participants were familiar with these concepts but did not necessarily start out that way. For example, in the first session P1 had trouble remembering to create functions by first providing an example call, but by the end of the study was doing so without any prompting from the facilitator. P1 also initially had trouble finding particular variable definitions on the screen but felt more comfortable by the second session. Near the end of the second session P2 expressed, "I want a let binding. . . I don't have any confidence I can make let bindings," despite having successfully done so many times by double-clicking the subcanvas or dragging values into the subcanvas. P2 instantly understood after a quick reminder from the facilitator, but it is notable that even after around three hours with the tool it had not quite sunk in that most TVs are let-bindings.

At the end of the study, we asked participants about writing expressions without naming them first. P1 expected to prefer being required to always provide a name; P2 was unsure, but noted that MANIPOSYNTH's default names had improved from the first version they used. In particular, at P2's behest we hard-coded the default names for list destruction to be `hd::tail` instead of the original type-based `a2::a_list2`. Even so, we rediscovered that naming was important in programming. Function skeletons are still inserted with generic parameter names, e.g. `fun x2 x1 -> (??)`, which are both unhelpful and backwards. This indeed resulted in user mistakes, and is a point to improve in future versions of MANIPOSYNTH.

Despite a few troubles, both participants were positive overall about the non-linear workflow. P1 noted the non-linear style "fits a lot more with how I like to write code," and P2 said, "I like it, I'm excited about it."

Table 2 User study participant interaction with synthesis, reporting the number of exercises using synthesis, the number of invocations of the synthesizer, the mean number of assertions and holes when invoked, and the usefulness of results: the percentage of invocations in which the synthesizer returned no result (timeout or crash), returned a result that was completely rejected by the user, and returned a result that was at least partially accepted by the user.

Participant	#Ex	#Synth	Mean #Assert	Mean #Hole	% No Result	% Useless Result	% Useful Result
P1	7	52	3.3	2.0	48%	35%	17%
P2	7	46	3.1	1.0	30%	54%	15%
Total	**14**	**96**	**3.2**	**1.6**	**40%**	**44%**	**16%**

RQ3. How do users interact with program synthesis? We introduced participants to the synthesizer in the second session, at which point it lacked a statistics model (instead enumerating terms by size) and did not offer "Accept / Reject" buttons (instead requiring the user to Undo upon undesired results); these were added for the final session. We wanted to learn how comfortable users were with providing assertions and using the synthesizer.

Participants were familiar with writing assertions. In the first session, the facilitator only introduced participants to providing example function calls, not asserting on their results. Despite this, unprompted, both participants wanted to make assertions once they had an example to work with. When assertions were formally introduced, participants were generally comfortable providing examples, although P1 would occasionally write assertions in a polymorphic form, e.g. `foldl f acc [] = acc`, which would insert new blank bindings for `f` and `acc` on the canvas and P1 would have to recover from the mistake. Even so, P1 appreciated that MANIPOSYNTH encouraged them to write in a test-driven development (TDD) style, and suspected it prevented them from making simple errors. When asked if they had trouble writing assertions, P2 responded, "I had trouble *not* making assertions," because P2 enjoyed toying with the synthesizer, but P2 did observe that constructing trees was a little tricky. MANIPOSYNTH currently only beautifies tree values, not tree expressions. Overall, when we asked how laborious it was to create examples on a scale of 1 to 10, P1 and P2 responded with 2 and 4, respectively. Providing assertions was not a bottleneck.

The facilitator introduced synthesis to the participants with the list `length` example, which left a positive first impression on the participants. Synthesis was somewhat less helpful after the `length` example. Overall, the participants invoked the synthesizer a total of 96 times (6.9 times per exercise), with only 16% of those invocations returning a result that the user partially or fully accepted (Table 2). Despite the low success rate, participants appreciated the synthesizer enough when it succeeded that they were not overly bothered when it did not, and were therefore comfortable invoking synthesis many times.

Before the addition of the "Accept / Reject" buttons in the third session there was also no feedback in MANIPOSYNTH that clearly indicated what had changed – P1 admitted to looking at their Vim window to ascertain what the synthesizer produced. The addition of the "Accept / Reject" interface was appreciated by participants and P1 noted they did keep their focus more on the MANIPOSYNTH window.

Overall, the facilitator's impression was that the participants were comfortable trying to use synthesis, but did not necessarily obtain mastery of it, in part because synthesis is opaque. P1 noted, "It is really hard to know whether synthesis is failing because I have posed the problem in an incorrect way or synthesis is failing because I haven't given it a lot of information. But the process of trying to give it more information is very illuminating in terms of whether my conception of the problem is wrong." P2 initially felt that working

with the synthesizer was unfamiliar but remained intrigued by its potential, saying, "It was kind of awkward at first. It sort of seemed like a cool trick but there were parts where it would actually complete the program which was kind of nice even though it was not like a very trivial program. That's a neat feature." These experiences suggest that synthesis in this setting is a viable workflow, despite its initial unfamiliarity.

RQ4. How do users interact with their text editor? Participants were allowed to use their text editor, but the heavy focus on learning Maniposynth meant that they only did so only as a last resort. P1 estimated they spent about 40% of their time looking at Vim when trying to figure out what was going on, but, by the end of the third session, only felt the need to edit in Vim on particularly tricky errors. P2 also felt more comfortable in Vim, "When I was really stuck, I felt self-conscious and I was like, "Alright I'll just figure this out in Vim quickly." It's faster, probably, I've got years of experience doing that."

Part of the promise of bimodal editing is that one *can* do this! Even so, Maniposynth may be over-reliant on only using shapes and colors to differentiate different kinds of elements, which may have driven the participants to look at their Vim window to understand what was happening instead of relying solely on the Maniposynth display.

SQ1. What are the pain points? How might the system be improved? Participants had trouble keeping track of all the elements on the display. Maniposynth relies on colors and shapes to distinguish the multitude of different UI elements: expressions, values, function parameters, assertions, expected values, return expressions, patterns, let-bindings (TVs), and different (sub)canvases that hold let-bindings. Both participants expressed a desire for more explicit labeling of what all these different elements were. After the first session, we added labels on the subcanvases ("Top level", "Bindings inside function", "Return expression(s) and value(s)") which P1 expressed appreciation for. We hoped those would obviate the need for more labeling, but at the end of the final session participants still desired clearer markings.

SQ2. Are participants comfortable enough to complete a task without guidance? After each exercise we asked participants if they felt comfortable completing the next task without assistance from the facilitator. By the end of the final session P2 was comfortable with minimal assistance, whereas P1 still felt the need for help. Although P1 understood the tool well, they still stumbled over different small issues such as UI corner cases and accidentally trying to edit a parent expression in the subexpression editor (discussed in SQ3 below).

SQ3. What else can we learn using the lenses of Cognitive Dimensions of Notations? This framework [25] comprises thirteen lenses for qualitatively assessing design trade-offs. Below, we report a subset of our observations from considering these lenses.

Diffuseness (How noisy is the display?) Maniposynth stores extra information, such as 2D binding coordinates and previously rejected synthesized expressions, as annotations in the OCaml code. Although Maniposynth includes a syntax highlighting rule that will gray out these AST annotations, the rule only works in VS Code with the Highlight extension [69] installed. P1 opined that, "All the annotations do make it less attractive to try to do stuff in Vim." Rejected expressions were particularly confusing because they appeared in their entirety in the code, albeit wrapped with `[@not ... ]`. Participants would sometimes read these large expressions thinking it was the code they were writing. After the study, we modified Maniposynth to instead store a short hash of the rejected expression.

Secondary Notation (Is there non-semantic notation to convey extra meaning?) Currently, Maniposynth does not support comments. P1 missed having comments, while P2 did not.

Viscosity (How hard is it to make changes?) Three main scenarios arose where changes were difficult. First, editing a base case requires that some execution hits the base case, otherwise the base case can never be focused; this was occasionally a hindrance and might be addressed either by adding a "phantom call frame" that focuses the case without a concrete execution or by automatically synthesizing an example that hits the case. Second, once an expression was in the program, it was hard to wrap the existing expression with some new expression; it would be better if there were a mechanism to indicate whether a new drag-and-dropped expression should replace or wrap the old. Finally, although subexpressions can be text-edited by double-clicking them on the display, only that subexpression is opened for editing. Sometimes participants (and the first author) would start editing a subexpression but realize they needed to edit a parent instead. We have since changed Maniposynth to open the entire parent for editing but with the clicked subexpression initially selected.

Visibility (Is everything needed visible? Can items be juxtaposed?) Element positioning in Maniposynth proved tricky, because elements will change size based on the size of the values in the TVs – multiple large trees in the function IO grid, for example, can make a function take up the whole window. Participants did have to move assertions around. P2 used a large screen and expected their functions to grow rightward: P2 would position assertions far to the right of their nascent function. P2 also expressed the desire for snap-to-grid so they could align their TVs perfectly. P1 used a smaller screen which may have caused trouble: at one point P1 was trying to debug and realized after-the-fact that they had scrolled the IO grid offscreen – had it been onscreen and they looked at it, they might have found their mistake quicker. One possible mitigation is to scale down large values.

5 Related Work

Several systems share our goal to center live program values in the programming workflow.

Programming by Demonstration (PBD). In this interaction paradigm, the user demonstrates an algorithm step-by-step, resulting in a program. The first PBD system, Pygmalion [67], targeted generic programming and, like Maniposynth, displayed the live values in scope as the object of user actions. For example, a function call with missing arguments was represented as an icon on the canvas. When all arguments were supplied, the icon was replaced with a display of its result value. To use that result value, the user dragged the value to where they wanted to use it. Recursion was supported. Although the 2D canvas was non-linear, Pygmalion treated the program as an imperative, step-by-step movie over time and did not offer a corresponding always-editable text representation.

Like Pygmalion, Pictorial Transformations (PT) [35] also offered program construction via step-by-step manipulation of live program values. PT allowed the user to customize visualizations, and was generally more expressive than Pygmalion, supporting more complicated algorithms including those involving lists. Later PBD systems were usually more domain-specific [13, 47], although ALVIS Live [37] targeted iterative array algorithms by demonstration, notably representing the resulting program in editable text. Unlike Maniposynth, ALVIS Live generated imperative code and could not offer non-linear editing – UI buttons were needed to allow users to move backwards and forwards in the timeline.

Some empirical evidence of benefits from a value-centric workflow was provided by the Pursuit PBD system for shell scripting [54]. In that work, a comic-strip style representation of a program – with before and after values in the frames of a comic-strip-enabled users

to more accurately generate programs compared to a more textual representation. On the other hand, when Frison [1] compared student performance between Python Tutor [29], providing editable code plus live output, and AlgoTouch [22], providing non-editable code plus PBD on values, they found students performed comparably in either environment. An analogous comparison in ALVIS Live also found similar overall student performance when using text or PBD [38]. These results can be interpreted either way: pessimistically, that value-centric manipulation is not clearly better; or optimistically, that despite non-editable code, value-centric editing performs as well as ordinary programming. Even so, a PBD environment may aid in avoiding initial fumbling with syntax and in discovering what a tool can do: Hundhausen et al. [38] found that on the first task, users in the PBD condition worked faster, more accurately, and spent less time consulting documentation.

Malleable Live Objects. In the object-oriented paradigm, the Self [72] language and environment displayed live objects graphically, allowing messages to be sent via direct manipulation (demonstrated in video form in [70]). Although value-centric, the interactions provided by Self and related systems, e.g. the Morphic UI framework [49], differ from all other systems discussed here in that manipulations in Self-like systems modify *state*, not the algorithm.

Like Self, the "Direct Programming" prototype [18] by Edwards allows users to directly invoke actions on displayed values, but, unlike Self, reified these actions in a script, blurring the line between running a program and modifying it. Also blurring the line between runtime interaction and coding, Boxer [15] was a non-linear programming environment displaying nested boxes on a 2D canvas. A box could contain a comment, code, a value, or serve as a graphics buffer for drawing. Boxes can be edited via code or by user interaction, enabling a workflow that mixes program runtime interaction with program creation. Boxer aimed for its interface to be an approachable computational medium, resulting in design choices that differ from MANIPOSYNTH. Boxer is not bimodal – the displayed boxes are the program – and state and code are mixed. Also, box results are not automatically rendered. Code boxes must be manually invoked and must write their results to another box, but Boxer includes mechanisms for configuring keys or mouse buttons to trigger particular boxes.

Live Nodes-and-Wires. In 2D nodes-and-wires programming [71], nodes usually represent transformations (expressions) and the wires represent dataflow (values). Consequently, nodes-and-wires environments do not necessarily display live values, although some systems do output live values below the nodes (e.g. natto.dev [65]). Among these environments, Enso [20], formerly known as Luna, is also bimodal like MANIPOSYNTH, offering both textual and graphical representations for editing the program.

PANE [34] flips the usual node-and-wires paradigm, instead using example values for nodes and locates transformations (expressions) on wires, placing values more at the center of attention compared to its peers. Example values can be clicked to invoke operations on them. PANE does not, however, maintain an editable text representation of the program.

Live Programming. Like MANIPOSYNTH, traditional live programming research seeks to augment ordinary, text-based coding with display of live program values, although the displayed values are read-only. There are a growing number of such systems. Python Tutor [29] is a popular teaching tool for visualizing program state in Python and other languages. Bret Victor's Inventing on Principle presentation [73] demonstrated several live programming environments and served as inspiration for later work [40, 46]. Edwards [17]

showed how examples can be incorporated into the IDE for live execution, and Babylonian-style Programming [61] explored how to better manage multiple examples – individual examples could be switched on and off, an interaction we could adopt in Maniposynth to selectively reduce the number of values shown in the function IO grids.

In-Editor PBE/PBD. Like our programming by example (PBE) synthesizer, recent work explores PBE and PBD interactions in textual environments. Several systems generate code within a computational notebook via manipulation of visualized values. Wrex [16] adapts the FlashFill [27] PBE workflow to Pandas dataframes in Jupyter notebooks – after demonstrating examples of a desired data transformation in a dataframe spreadsheet view, Wrex outputs readable Python code. Similarly, the PBD systems B2 [75], mage [41], and Mito [14] transform step-by-step interactions on displayed notebook values into Python code. For a Haskell notebook environment, Vital [30, 31] offers copy-paste operations on visualized algebraic data type (ADT) values, which are realized by changing the textual code in the appropriate cell. Like Maniposynth, graphical interactions in Vital can extract subvalues via pattern matching, although Vital's workflow focuses on modifying single values in place rather than building up computations like in Maniposynth. While these notebook systems provide some manipulation of intermediate values, none offer fine-grained non-linearity.

For more ordinary IDE settings, CodeHint [24], SnipPy [21], and JDial [36] provide synthesis interactions in the live context of the user's incomplete code. With CodeHint, users set a breakpoint in their Java program and describe a property about a desired value – CodeHint enumerates method calls in the execution environment at the breakpoint to find a satisfying expression. Like Maniposynth, CodeHint uses a statistics model to rank results. Notably, users with CodeHint were significantly faster and more successful at completing given tasks than users without. For Python, SnipPy [21] adapts the Projection Boxes tabular display of runtime values [46] to perform PBE in the context of live Python values. The authors validated that users successfully used the synthesizer to complete portions of given tasks. JDial [36] records variable values during execution of an imperative Java program and allows the programmer to directly change incorrect values in the trace. The program is repaired to match the corrections via sketch-based synthesis [68]. JDial, however, is limited to small program repairs and does not offer program construction features.

Bidirectional, Bimodal Programming. Some systems represent programs as ordinary text, but also allow direct manipulation on program *outputs* to be back-propagated to change the original code. Usually, these changes are "small" changes to literals in the program – such as numbers [11, 43, 50, 23], strings [74, 64, 43, 50], or lists [50]. More full-featured program construction via output manipulations is available in a few systems for programs that output graphics, such as APX [51], Transmorphic [63], and Sketch-n-Sketch [33].

Although Maniposynth also centers values as subjects for manipulation, we do not yet apply bidirectional techniques to deeply back-propagate a change on a value – direct changes on a value are only allowed when it was introduced as a literal in the immediately associated expression. An earlier version of Maniposynth did support limited back-propagation, which we disabled because it caused trouble in the user study: manipulating a value would inconspicuously change a literal in a very different part of the program. Determining an understandable meaning of such direct changes on a value remains an avenue for future work. While the bidirectional programming community offers the "least change" principle [52], that minimal changes should be performed to maintain a constraint, in the context of a

full program a change may cause confusion not because of its magnitude but because the item changed is far from the user's focus. Revealing that far-away code by popping open a "bubble" [7, 8] or "portal" [9] may be one way to help make the change understandable.

6 Future Work and Conclusion

Scaling Up. So far, Maniposynth has been applied only to small, side-effect-free programs. Running larger programs requires managing traces efficiently, and handling practical programs requires managing side effects such as input/output and mutation.

Tracing has the potential to consume a considerable amount of memory: every execution step produces a new portion of the trace. Currently, traces are all stored in RAM. Future tracing could instead write to persistent storage. Furthermore, for a large program, only a small portion of the trace will likely be needed at a time. Tracing can be skipped for code outside the region of interest. When needed, missing portions of the trace could be rebuilt on demand via program replay, which can be accomplished efficiently by periodically dumping the program state during the initial execution and replaying from a checkpoint as needed [5].

Support for side effects requires both changes to the UI in addition to technical engineering. Side effects are necessarily linear, whereas Maniposynth's UI is currently based around a free-form, unordered 2D canvas. One UI possibility is to have each imperative statement spatially divide its (sub)canvas into "before" and "after" – TVs spatially above the imperative statement are executed before it, and TVs below are executed afterwards. For the implementation, careful interception and logging of system calls can record all program I/O and enable deterministic replay, even for large programs [57]. In Maniposynth, recording and replaying side effects could be handled in the interpreter rather than at the system call level.

Value-Oriented Thinking. We hypothesize that expression-oriented and value-oriented modes of thinking are distinct states of mind, and experienced programmers tend towards the former. An intriguing possibility for future work is to experimentally validate that expression-oriented and value-oriented thinking are actually modes – i.e. the activity of considering values discourages considering expressions, and vice versa. More immediately, there are possible changes to Maniposynth that might encourage more value-focused interaction.

One experiment is to change the display of variable uses so that, instead of the name of the variable, its current value is shown instead, with the name as a tooltip or subscript. This change might nudge users out of the expression-oriented mode of thinking back towards value-oriented thinking. An intriguing corollary experiment was requested by P1. To keep track of their provenance, P1 wanted values to be drawn with unique colors all the time, rather than only when autocomplete menus were open. Another possibility is, when the cursor is over a variable usage, to highlight the TV where the variable is defined. We would like to explore these display choices, as well as other opportunities for "linked" visualizations [60].

Finally, while dragging items onto an expression is quite useful, dragging items onto values is currently less so. When working through the examples, the implementer dragged some item onto a value on only 4 occasions, compared to dragging onto an expression 209 times. In the future, dragging a value to a value might open a menu of possible ways to combine the values. Ideally, programmers should be able to customize the available actions, as in Vital [31] which includes an API for this purpose.

Maniposynth currently focuses on interactions on relatively small values. Larger data sets might be more more conveniently displayed and manipulated in a spreadsheet-style view – Flowsheets [10] demonstrates one such approach, albeit without program synthesis.

Conclusion. How close is Maniposynth to achieving its goals of providing a value-centric, non-linear programming environment? Based on the examples we implemented and feedback from our study participants, Maniposynth largely succeeded at providing useful live values. The non-linear features functioned moderately well – users rarely had to think about binding order – but Maniposynth was not immediately learnable and would benefit from more explicit labeling of the various kinds of elements on the canvas.

Overall, building on the insight from Eros [19] that non-linearity complements functional programming, Maniposynth shows that non-linearity can be maintained even when the program is ordinary code. Our emphasis on textual code has resulted in Maniposynth currently being somewhat more expression-centric than Eros. As described above, there are many possible ways Maniposynth might become more value-centric, to further our vision to make programming feel like a tangible process of molding and forming.

References

1. Michel Adam, Moncef Daoud, and Patrice Frison. Direct Manipulation versus Text-based Programming: An Experiment Report. In *Conference on Innovation and Technology in Computer Science Education (ITiCS)*, 2019.
2. Haytham Amairah. Wrong Pattern for Filling Month Names. `https://techcommunity.microsoft.com/t5/excel/flash-fill-wrong-pattern-for-filling-month-names/m-p/355213`, 2019.
3. Andrew Begel. LogoBlocks: A Graphical Programming Language for Interacting with the World, 1996. Advanced Undergraduate Project, MIT Media Lab.
4. Josh Berdine, Guillaume Petiot, hhugo, and contributors. Ocamlformat. `https://github.com/ocaml-ppx/ocamlformat`, 2021.
5. Bob Boothe. Efficient Algorithms for Bidirectional Debugging. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000. `doi:10.1145/349299.349339`.
6. Frédéric Bour, Thomas Refis, Gemma Gordon, Simon Castellan, and contributors. Merlin. `https://github.com/ocaml/merlin`, 2021.
7. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *International Conference on Software Engineering (ICSE)*, 2010.
8. Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Conference on Human Factors in Computing Systems (CHI)*, 2010.
9. Alexander Breckel and Matthia
In *International Conference on*
`2016.7503732`.
10. Glen Chiacchieri. F-F-F-Flows 🎉 demo. `https://www.youtube.com/watch?v=y1Ca5czOY7Q`, 2017.
11. Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and Direct Manipulation, Together at Last. In *Conference on Programming Language Design and Implementation (PLDI)*, 2016.
12. Naëla Courant, Julien Lepiller, and Gabriel Scherer. camlboot. `https://github.com/Ekdohibs/camlboot/`, 2020.
13. Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration.* MIT Press, 1993.
14. Jacob Diamond-Reivich. Mito: Edit a Spreadsheet. Generate Production Ready Python. In *LIVE Workshop*, 2020.

15 Andrea A. diSessa and Harold Abelson. Boxer: A Reconstructible Computational Medium. *Communications of the ACM (CACM)*, 1986.
16 Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A Unifed Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. *Conference on Human Factors in Computing Systems (CHI)*, 2020.
17 Jonathan Edwards. Example Centric Programming. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2004. `doi:10.1145/1028664.1028713`.
18 Jonathan Edwards. Direct Programming. `https://vimeo.com/274771188`, 2018.
19 Conal Elliott. Tangible Functional Programming. In *International Conference on Functional Programming (ICFP)*, 2007. URL: `http://conal.net/papers/Eros/`.
20 Enso. Enso. `https://enso.org/`.
21 Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-Step Live Programming by Example. In *Symposium on User Interface Software and Technology (UIST)*, 2020.
22 Patrice Frison. A Teaching Assistant for Algorithm Construction. In *Conference on Innovation and Technology in Computer Science Education (ITiCS)*, 2015.
23 Koumei Fukahori, Daisuke Sakamoto, Jun Kato, and Takeo Igarashi. CapStudio: An Interactive Screencast for Visual Application Development. In *Conference on Human Factors in Computing Systems (CHI), Extended Abstracts*, 2014.
24 Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *International Conference on Software Engineering (ICSE)*, 2014.
25 Thomas R. G. Green and Marian Petre. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *J. Vis. Lang. Comput.*, 1996. `doi:10.1006/jvlc.1996.0009`.
26 Rudi Grinberg, Andrey Popp, Rusty Key, Louis Roché, Oleksiy Golovko, Sacha Ayoun, cannorin, Ulugbek Abdullaev, Thibaut Mattio, and Max Lantas. OCaml-LSP. `https://github.com/ocaml/ocaml-lsp`, 2021.
27 Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Symposium on Principles of Programming Languages (POPL)*, 2011.
28 Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program Synthesis. *Foundations and Trends in Programming Languages*, 2017. `doi:10.1561/2500000010`.
29 Philip J. Guo. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Technical Symposium on Computer Science Education (SIGCSE)*, 2013.
30 Keith Hanna. Interactive Visual Functional Programming. In *International Conference on Functional Programming (ICFP)*, 2002.
31 Keith Hanna. A Document-Centered Environment for Haskell. In *International Workshop on Implementation and Application of Functional Languages (IFL)*, 2005.
32 Brian Hempel. The Magnificent Maniposynth. `http://maniposynth.org`, 2022.
33 Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Symposium on User Interface Software and Technology (UIST)*, 2019.
34 Joshua Horowitz. PANE: Programming with Visible Data. In *LIVE Workshop*, 2018. URL: `http://joshuahhh.com/projects/pane/`.
35 Yen-Teh Hsia and Allen L. Ambler. Programming Through Pictorial Transformations. In *International Conference on Computer Languages*, 1988.
36 Qinheping Hu, Roopsha Samanta, Rishabh Singh, and Loris D'Antoni. Direct Manipulation for Imperative Programs. In *Static Analysis Symposium (SAS)*, 2019.
37 Christopher D. Hundhausen and Jonathan Lee Brown. What You See Is What You Code: A live Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing*, 2007. `doi:10.1016/j.jvlc.2006.03.002`.

38 Christopher D. Hundhausen, Sean Farley, and Jonathan Lee Brown. Can Direct Manipulation Lower the Barriers To Computer Programming and Promote Transfer of Training? An Experimental Study. *ACM Trans. Comput. Hum. Interact.*, 2009. doi:10.1145/1592440.1592442.

39 Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. Guiding Dynamic Programing Via Structural Probability for Accelerating Programming by Example. *Proc. ACM Program. Lang.*, (OOPSLA), 2020. doi:10.1145/3428292.

40 Saketh Kasibatla and Alex Warth. Seymour: Live Programming for the Classroom. In *LIVE Workshop*, 2017.

41 Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid Moves Between Code and Graphical Work In Computational Notebooks. In *Symposium on User Interface Software and Technology (UIST)*, 2020.

42 Andrew J. Ko and Brad A. Myers. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Human Factors in Computing Systems (CHI)*, 2006.

43 Kevin Kwok and Guillermo Webster. Carbide Alpha, 2016. URL: https://alpha.trycarbide.com/.

44 A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 1960.

45 Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating Search-based Program Synthesis using Learned Probabilistic Models. In *Conference on Programming Language Design and Implementation (PLDI)*, 2018.

46 Sorin Lerner. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. *Conference on Human Factors in Computing Systems (CHI)*, 2020.

47 H. Lieberman, editor. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann Publishers Inc., 2001.

48 Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.*, (ICFP), 2020. doi:10.1145/3408991.

49 John H. Maloney and Randall B. Smith. Directness and Liveness In the Morphic User Interface Construction Environment. In *Symposium on User Interface Software and Technology (UIST)*, 1995.

50 Mikaël Mayer, Viktor Kunčak, and Ravi Chugh. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2018.

51 Sean McDirmid. A Live Programming Experience. In *Future Programming Workshop, Strange Loop*, 2015. https://www.youtube.com/watch?v=YLrdhFEAiqo. URL: https://onedrive.live.com/download?cid=51C4267D41507773&resid=51C4267D41507773%2111492&authkey=AMwcxdryTyPiuW8.

52 Lambert Meertens. Designing Constraint Maintainers for User Interaction. https://www.kestrel.edu/people/meertens/pub/dcm.pdf, 1998.

53 Microsoft. Visual studio code. https://code.visualstudio.com, 2022.

54 Francesmary Modugno, Albert T. Corbett, and Brad A. Myers. Graphical Representation of Programs In a Demonstrational Visual Shell - An Empirical Evaluation. *ACM Trans. Comput. Hum. Interact.*, 1997. doi:10.1145/264645.264659.

55 National Instruments. Labview. URL: https://www.ni.com/en-us/shop/labview.html.

56 Tobias Nipkow and Mohammad Abdulaziz. Functional Data Structures (in2347). https://github.com/nipkow/fds_ss20/tree/daae0f92277b0df86f34ec747c7b3f1c5f0a725c, 2020. Technische Universität München.

57 Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *USENIX Annual Technical Conference*, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan.

58 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live Functional Programming with Typed Holes. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.

59 Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

60 Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. Linked Visualisations Via Galois Dependencies. *Proc. ACM Program. Lang.*, (POPL), 2022. `doi:10.1145/3498668`.

61 David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style Programming - Design and Implementation of An Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.*, 2019. `doi:10.22152/programming-journal.org/2019/3/9`.

62 Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Communications of the ACM (CACM)*, 2009.

63 Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. *Transmorphic: Mapping Direct Manipulation to Source Code Transformations*. HPI/Potsdam University, 2017.

64 Christopher Schuster and Cormac Flanagan. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. In *LIVE Workshop*, 2016.

65 Paul Shen. natto.dev. `https://natto.dev/`.

66 Ben Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, August 1983.

67 David Canfield Smith. *Pygmalion: A Creative Programming Environment*. PhD thesis, Stanford University, 1975.

68 Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

69 Fabio Spampinato. Highlight VS Code Extension. `https://marketplace.visualstudio.com/items?itemName=fabiospampinato.vscode-highlight`, 2021.

70 Sun Microsystems. Self: The movie;. `http://www.smalltalk.org.br/movies/self.html`, 1995.

71 William Robert Sutherland. *The On-line Graphical Specification of Computer Procedures*. PhD thesis, Massachusetts Institute of Technology, 1966.

72 David M. Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 1987.

73 Victor, Bret. Inventing on Principle, 2012. URL: `https://vimeo.com/36579366`.

74 Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

75 Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. B2: Bridging Code and Interactive Visualization In Computational Notebooks. In *Symposium on User Interface Software and Technology (UIST)*, 2020.

Synchron – An API and Runtime for Embedded Systems

Abhiroop Sarkar ✉
Chalmers University of Technology, Gothenburg, Sweden

Bo Joel Svensson ✉
Chalmers University of Technology, Gothenburg, Sweden

Mary Sheeran ✉
Chalmers University of Technology, Gothenburg, Sweden

Abstract

Programming embedded applications involves writing concurrent, event-driven and timing-aware programs. Traditionally, such programs are written in machine-oriented programming languages like C or Assembly. We present an alternative by introducing Synchron, an API that offers high-level abstractions to the programmer while supporting the low-level infrastructure in an associated runtime system and one-time-effort drivers.

Embedded systems applications exhibit the general characteristics of being (i) concurrent, (ii) I/O–bound and (iii) timing-aware. To address each of these concerns, the Synchron API consists of three components – (1) a Concurrent ML (CML) inspired message-passing concurrency model, (2) a message-passing–based I/O interface that translates between low-level interrupt based and memory-mapped peripherals, and (3) a timing operator, `syncT`, that marries CML's `sync` operator with timing windows inspired from the TinyTimber kernel.

We implement the Synchron API as the bytecode instructions of a virtual machine called SynchronVM. SynchronVM hosts a Caml-inspired functional language as its frontend language, and the backend of the VM supports the STM32F4 and NRF52 microcontrollers, with RAM in the order of hundreds of kilobytes. We illustrate the expressiveness of the Synchron API by showing examples of expressing state machines commonly found in embedded systems. The timing functionality is demonstrated through a music programming exercise. Finally, we provide benchmarks on the response time, jitter rates, memory, and power usage of the SynchronVM.

2012 ACM Subject Classification Computer systems organization → Embedded software; Software and its engineering → Runtime environments; Computer systems organization → Real-time languages; Software and its engineering → Concurrent programming languages

Keywords and phrases real-time, concurrency, functional programming, runtime, virtual machine

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.17

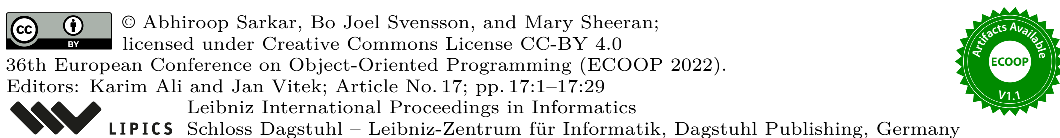
Related Version *Full Version*: `https://tinyurl.com/ymert8v2`

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.25`
archived at `swh:1:dir:6d5121ddfdfcec8add38fae75de4d51c6b194690`

Funding This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and by the Chalmers Gender Initiative for Excellence.

1 Introduction

Embedded systems are ubiquitous. They are pervasively found in application areas such as the internet of things, industrial machinery, automobiles, robotics, etc. Embedded systems applications tend to embody three common characteristics:


© Abhiroop Sarkar, Bo Joel Svensson, and Mary Sheeran;
licensed under Creative Commons License CC-BY 4.0

1. They are *concurrent* in nature.
2. They are predominantly *I/O–bound* applications.
3. A large subset of such applications are *timing-aware*.

Programming applications with the above characteristics involves low-level hardware interactions via callback-based driver APIs. Such programs tend to be expressed in low-level languages like C in the form of complex state machines, which often results in difficult-to-maintain and elaborate state-transition tables. Moreover, C programmers use error-prone shared-memory primitives like *semaphores* and *locks* to mediate interactions that occur between the callback-based driver handlers.

In modern microcontroller runtimes, like MicroPython [12] and Espruino (Javascript) [37], higher-order functions can be used to handle callback-based APIs. However, a common pitfall in these languages is the chaining of nested callback handlers, which leads to a form of accidental complexity known as *callback-hell* [20]. Such programs have a non-linear control flow and are difficult to express, read and maintain.

We present Synchron, an API and accompanying runtime that aims to address the above concerns about callback-hell and shared-memory concurrency while targeting the three earlier mentioned characteristics of embedded programs by a combination of:

1. A message-passing–based concurrency model inspired from Concurrent ML.
2. A message-passing–based I/O interface that unifies concurrency and I/O.
3. A notion of time that fits the message-passing concurrency model.

Concurrent ML (CML) [25] builds upon the synchronous message-passing–based concurrency model CSP [16] but adds the feature of composable first-class *events*. Events allow the programmer to tailor new concurrency abstractions and express application-specific protocols. Moreover, a synchronous concurrency model renders linear control-flow to a program, as opposed to bottom-up, non-linear control flow exhibited by asynchronous callback APIs.

Synchron extends CML's message-passing API for software processes to I/O and hardware interactions by modelling the external world as a process through the `spawnExternal` operator. As a result, the standard message-passing functions such as `send, receive` etc. become usable for I/O interactions, such as asynchronous driver interrupts. The overall API design allows efficient scheduling and limited power usage of programs via an associated runtime.

For timing, Synchron introduces the `syncT` operator that allows the specification of baseline and deadline times for communication between message-passing processes. The logical timing model endowed by this operator helps the prevention of *jitter* associated with the execution of real-time, periodic applications.

The Synchron API is implemented in the form of a bytecode-interpreted virtual machine (VM) called SynchronVM. Internally, the SynchronVM runtime manages the scheduling and timing of the various processes, interrupt handling, memory management, and other bookkeeping infrastructure. Notably, the runtime system features a low-level bridge interface that translates low-level hardware interrupts or memory-mapped I/O into software messages, enabling application-level processes to use the message-passing API for low-level I/O.

Contributions

- We identify three characteristic behaviours of embedded applications, namely being (i) concurrent, (ii) I/O–bound, and (iii) timing-aware, and propose a combination of abstractions, the Synchron API (Section 3), that address these requirements.
- **Message-passing–based I/O.** Synchron's message-passing API combines *concurrency* and *callback–based I/O* to a single interface. A software message or a hardware interrupt is identical in the Synchron API, providing the programmer with a simpler message-based framework to express concurrent hardware interactions. We show the I/O API in Section 3.2 and describe the core runtime algorithms to support this API in Section 4.

- **Declarative state machines for embedded systems.** Combining CML primitives with our I/O interface presents a declarative framework to express state machines. We illustrate this through case studies using the Synchron API in Sections 6.1 and 6.2.
- **Evaluation.** We implement the Synchron API and its associated runtime within a virtual machine, SynchronVM, described in Section 5. We illustrate the practicality and expressivity of our API by presenting three case studies in Section 6 that runs on the STM32 and NRF52 microcontroller boards. Finally, we show response time, memory and power usage, jitter rates, and load testing benchmarks on the SynchronVM in Section 7.

2 Motivation

Concurrency and I/O. Embedded system applications are primarily I/O-bound in nature. The low-level I/O-interface provided by hardware drivers is typically callback-based. The callback style of programming is complicated but offers benefits when it comes to energy efficiency. Registering a callback with an Interrupt Service Routine (ISR) allows the processor to go to sleep and conserve power until the interrupt arrives.

Callback-based programming renders a non-linear control flow to embedded system programs, which is best handled via concurrent threads as provided by ZephyrOS, ChibiOS or FreeRTOS. These programs tend to be error-prone and hard to maintain. The problems are compounded by the fact that C is not an intrinsically concurrent language and handles concurrency through ad-hoc language extensions.

Time. A close relative of concurrent programming for embedded systems is *real-time programming*. Embedded systems applications such as digital sound cards routinely exhibit behaviour where the time of completion of an operation determines the correctness of the program. C handles real-time applications via the underlying OS APIs.

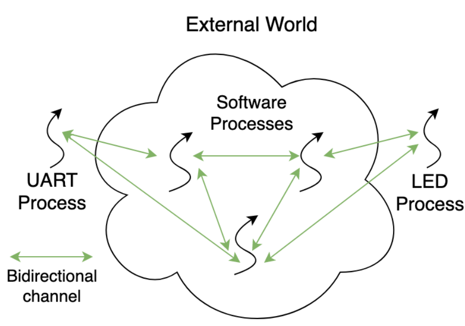
Typical real-time APIs support prioritised threads. For instance, the `FreeRTOS Task` API allows a programmer to define a static *priority* number for each thread. As the number of concurrent threads grows, a limited range of priority numbers (1 – 5) results in clashes in thread priorities. Another common risk with priority-based systems is to run into the *priority inversion problem* [31], which can have fatal consequences in hard real-time scenarios. On the other hand, high-level language platforms for embedded systems (such as MicroPython [12]) typically lack native language support for timing-aware computations.

Problem Statement. We believe there exists a gap for a high-level language that can express concurrent, I/O–bound, and timing-aware programs for programming resource-constrained embedded systems. We outline our key idea to address this gap below.

2.1 Key Ideas

Our key idea is the Synchron API, which adopts a synchronous message-passing concurrency model and extends the message-passing functionality to all I/O interactions. Synchron also introduces *baselines* and *deadlines* for the message-passing, which consequently brings in a notion of time into the API. The resultant API is a collection of nine operations that can express (i) concurrency, (ii) I/O, and (iii) timing in a uniform and declarative manner.

The external world as processes. The Synchron API models all external drivers as processes that can communicate with the software layer through message-passing. Synchron's `spawnExternal` operator treats an I/O peripheral as a process and a hardware interrupt as a message from the corresponding process. Fig. 1 illustrates the broad idea.



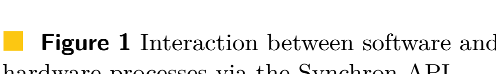


Figure 1 Interaction between software and hardware processes via the Synchron API.

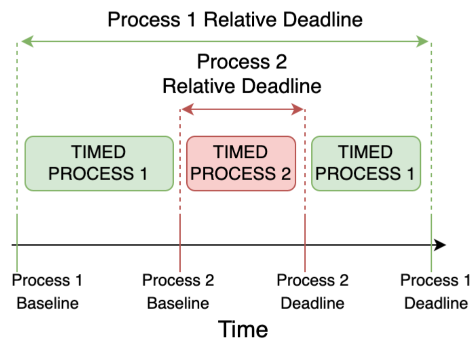


Figure 2 Dynamic priority change when using the `syncT` operation.

This design relieves the programmer from writing complex callback handlers to deal with asynchronous interrupts. The synchronous message-passing–based I/O renders a linear control-flow to I/O-bound embedded-system programs, allowing the modelling of state machines in a declarative manner. Additionally, the message-passing framework simplifies the hazards of concurrent programming with shared-memory primitives (like `FreeRTOS semaphores`) and the associated perils of maintaining intricate locking protocols.

Hardware-Software Bridge. The Synchron runtime enables the seamless translation between software messages and hardware interrupts. The runtime does hardware interactions through a low-level software *bridge* interface, which is implemented atop the drivers supplied by an OS like Zephyr/ChibiOS. The *bridge* layer serialises all hardware interrupts into the format of a software message, thereby providing a uniform message-passing interaction style for both software and hardware messages.

Timing. The final key component of the Synchron API is the real-time function, `syncT`, that instead of using a static priority for a thread (like Ada, RT-Java, FreeRTOS, etc.), borrows the concept of a dynamic priority specification from TinyTimber [19].

The `syncT` function specifies a *timing window* by stating the baseline and deadline of message communication between processes. Synchron's timing model assumes that computation takes zero time, so the time required for communication determines the timing window of execution of the entire process. As the deadline of a process draws near, the Synchron runtime can choose to dynamically change the priority of a process while it is running. Fig. 2 illustrates the idea of dynamic priority-change where a scheduler can choose to prioritise a second process over a running, *timed* process, although the running process has a deadline in the future. The preemptive Synchron runtime enables this API.

The combination of `syncT`, `spawnExternal` and the CML-inspired synchronous message-passing concurrency model constitutes the Synchron API that allows declarative specification of embedded applications. We suggest that this API is an improvement, in terms of expressivity, over the currently existing languages and libraries on embedded systems and provide illustrative examples to support this in Section 6. We also provide benchmarks on the Synchron runtime in Section 7. Next, we discuss the Synchron API in more detail.

3 The Synchron API

3.1 Synchronous Message-Passing and Events

Synchron adopts the synchronous message-passing API of Concurrent ML (CML) [25] –

```
spawn   : (() -> ()) -> ThreadId
channel : ()  -> Channel a
send    : Channel a -> a -> Event ()
recv    : Channel a -> Event a
sync    : Event a   -> a
```

In the above type signatures, the type parameter, `a`, indicates a polymorphic type. The call to `spawn` allows the creation of a new process whose body is represented by the $(() \to ())$ type. The `channel ()` call creates a blocking *channel* along which a process can send or receive messages. A channel blocks until a sender has a corresponding receiver and vice-versa.

Notable in the above API is the value of type `Event` returned by the `send` and `recv` calls. The central idea of an event is to break the act of synchronous communication into two steps:

(i) Expressing the intent of communication as an *event*-value

(ii) Synchronously communicating between the sender and receiver via the *event*-value

The first step above, accomplished through `send` and `recv`, results in the creation of a type of value called an `Event`. An *event* is a first-class value in the language that represents deferred communication. Reppy describes events as "first-class synchronous operations" [25]. Given a value of type `Event`, the second step of synchronising between processes and the consequent act of communication is accomplished via the `sync : Event a -> a` operation.

Furthermore, CML provides the `choose` operator to express multi-party communication (type signature shown below). The semantics of `choose` involves racing between two or more synchronous operations and *choosing* the one that succeeds first. Further compositional combinators, like `wrap`, enable higher-order composition of events.

```
choose : Event a → Event a  → Event a
wrap   : Event a → (a → b) → Event b
```

The advantage of representing communication as a first-class value is that event-based combinators can be used to build more elaborate communication protocols. For instance -

```
protocol : Event ()
protocol =
  choose (send c1 msg1) (wrap (recv c2) (λ msg2 -> sync (send c3 msg2)))
```

Using events, the above protocol involving multiple *sends* and *receives* was expressible as a procedural abstraction while still having the return type of `Event ()`. A consumer of the above protocol can further use the nondeterministic choice operator, `choose`, and *choose* among multiple protocols. This combination of a composable programming style and multiprocess program design allows this API to represent callback-based, state machine oriented programs in a declarative manner.

Comparisons between Events and Futures. The fundamental difference between events and futures is that of *deferred communication* and *deferred computation* respectively. A future aids in asynchronous computations by encapsulating a computation whose value is made available at a future time. On the other hand, an event represents deferred communication as a first-class entity in a language. Using the `wrap` combinator, it is possible to chain lambda functions capturing computations that should happen post-communication as well. However, events are fundamentally building blocks for communication protocols.

3.2 Input and Output

In the Synchron API, I/O is expressed using the same events as are used for inter-process communication. Each I/O device is connected to the running program using a primitive we call `spawnExternal` as a hint that the programmer can think of, for example, an LED as a process that can receive messages along a channel. Each *external* process denotes an underlying I/O device that is limited to send and receive messages along one channel.

```
1 spawnExternal : Channel a -> Driver -> ExternalThreadId
```

The first parameter supplied to `spawnExternal` is a designated fixed channel along which the external process shall communicate. The second argument requires some form of identifier to uniquely identify the driver. This identifier for a driver tends to be architecture-dependent. For instance, when using low-level memory-mapped I/O, reads or writes to a memory address are used to communicate with a peripheral. So the unique memory address would be an identifier in that case. On the other hand, certain real-time operating systems (such as `FreeRTOS` or `Zephyr`) can provide more high-level abstractions over a memory address. In the Synchron runtime, we number each peripheral in a monotonically increasing order, starting from 0. So our `spawnExternal` API becomes:

```
1 type DriverNo = Int
2 spawnExternal : Channel a -> DriverNo -> ExternalThreadId
```

To demonstrate the I/O API in the context of asynchronous drivers, we present a standard example of the *button-blinky* program. The program matches a button state to an LED so that when the button is down, the LED is on, otherwise the LED is off:

Listing 1 Button-Blinky using the Synchron API.

```
1  butchan = channel ()
2  ledchan = channel ()
3
4  glowled i = sync (send ledchan i)
5
6  f : ()
7  f = let _ = sync (wrap (recv butchan) glowled) in f
8
9  main = let _ = spawnExternal butchan 0 in
10        let _ = spawnExternal ledchan 1 in f
```

Listing 1 above spawns two hardware processes – an LED process and a button process. It then calls the function `f` which arrives at line 7 and waits for a button press. During the waiting period, the scheduler can put the process to sleep to save power. When the button interrupt arrives, the Synchron runtime converts the hardware interrupt to a software message and wakes up process `f`. It then calls the `glowled` function on line 4 that sends a switch-on message to the LED process and recursively calls `f` infinitely.

The above program represents an asynchronous, callback-based application in an entirely synchronous framework. The same application written in C, on top of the Zephyr OS, is more than 100 lines of callback-based code [11]. A notable aspect of the above program is the lack of any non-linear callback-handling mechanism. Aside from abstracting away the interrupt-handling mechanism, the program is highly extensible; adding a new interrupt handler is as simple as defining a new function.

3.3 Programming with Time

In a real-time scenario, a programmer wants to precisely control the response-time of certain operations. So the natural intuition for real-time C-extensions like FreeRTOS *Tasks* or languages like Ada is to delegate the scheduling control to the programmer by allowing them to attach a priority level to each process.

The priority levels involved decide how a tie is broken by the scheduler. However, with a small fixed number of priority levels, it is likely for several processes to end up with the same priority, leading the scheduler to order them fairly again within each level.

Another complication that crops up in the context of priorities is the *priority inversion problem* [31]. Priority inversion is a form of resource contention where a high-priority thread gets blocked on a resource held by a low-priority thread, thus allowing a medium priority thread to take advantage of the situation and get scheduled first. The outcome of this scenario is that the high-priority thread gets to run after the medium-priority thread, leading to possible program failures.

The Synchron API admits the *dynamic* prioritisation of processes, drawing inspiration from the TinyTimber kernel [21]. TinyTimber specifies a *timing window* as a baseline and deadline time, and a scheduler can use this timing window to determine the runtime priority of a process. The timing window expresses the programmer's wish that the operation is started at the *earliest* on the baseline and *no later* than the deadline.

In Synchron, a programmer specifies a *timing window* (of the wall-clock time) during which they want message synchronisation, that is the rendezvous between message sender and receiver, to happen. We do this with the help of the timed-synchronisation operator, `syncT`, with the type signature `syncT : Time -> Time -> Event a -> a`.

Comparing the type signature of `syncT` with that of `sync` :

```
1 syncT : Time -> Time -> Event a -> a
2 sync  :                 Event a -> a
```

The two extra arguments to `syncT` specify a lower and upper bound on the *time of synchronisation* of an event. The two arguments to `syncT`, of type `Time`, express the relative times calculated from the current wall-clock time. The first argument represents the *relative baseline* – the earliest time instant from which the event synchronisation should begin. The second argument specifies the *relative deadline*, i.e. the latest time instant (starting from the baseline), by which the synchronisation should start. For instance,

```
1 syncT (msec 50) (msec 20) timed_ev
```

means that the event, *timed_ev*, should begin synchronisation at the earliest 50 milliseconds and the latest $50 + 20$ milliseconds from *now*. The *now* concept is based on a thread's local view of what time it is. This thread-local time (T_{local}) is always less than or equal to wall-clock time ($T_{absolute}$). When a thread is spawned, its thread-local time, T_{local}, is set to the wall-clock time, $T_{absolute}$.

While a thread is running, its local time is frozen and unchanged until the thread executes a timed synchronisation, a `syncT` operation where time progresses to $T_{local} + baseline$.

```
1 process1 _ =
2   let _ = s1 in -- T_local = 0
3   let _ = s2 in -- T_local = 0
4   let _ = syncT (msec 50) (usec 10) ev1 in
5   process1 () -- T_local = 50 msec
```

The above illustrates that the *untimed* operations `s1` and `s2` have no impact on a thread's view of what time it is. In essence, these operations are considered to take no time, which is a reference to *logical* time and not the physical time. Synchron shares this logical timing model with other systems such as the synchronous languages [8] and ChucK [35].

In practice, this assumption helps control *jitter* in the timing as long as the timing windows specified on the synchronisation are large enough to contain the execution time of `s1`, `s2`, the synchronisation step and the recursive call. Local and absolute time must meet up at key points for this approach to work. Without the two notions of time meeting, local time would drift away from absolute time in an unbounded fashion. For a practical implementation of `syncT`, a scheduler needs to meet the following requirements:

- The scheduler should provide a mechanism for overriding fair scheduling.
- The scheduler must have access to a wall-clock time source.
- A scheduler should attempt to schedule synchronisation such that local time meets up with absolute time at that instant.

We shall revisit these requirements in Section 5 when describing the scheduler within the Synchron runtime. Next, we shall look at a simple example use of `syncT`.

Blinky

The well-known *blinky* example, shown in Listing 2, involves blinking an LED on and off at a certain frequency. Here we blink once every second.

Listing 2 Blinky using the `syncT` operation.

```
ledchan = channel ()

sec  n = n * 1000000
usec n = n -- the unit-time in the Synchron runtime

foo : Int -> ()
foo val =
 let _ = syncT (sec 1) (usec 1) (send ledchan val) in
 foo (not val) -- not flips 1 to 0 and 0 to 1

main = let _ = spawnExternal ledchan 1 in foo 1
```

In the above program, `foo` is the only software process; a single external hardware process for the LED driver communicates via the ledChan channel. Line 8 is the critical part of the logic that sets the period of the program at 1 second, and the recursion at Line 9 keeps the program alive forever. We discuss a more involved example using `syncT` in Section 6.3.

4 Synchronisation Algorithms

The synchronous nature of message-passing is the foundation of the Synchron API. In this section, we describe the runtime algorithms, in an abstract form, that enable processes to synchronise. The Synchron runtime implements these algorithms, which drives the scheduling of the various software processes.

In Synchron, we synchronise on events. **Events**, in our API, fall into the categories of *base* events and *composite* events. The base events are `send` and `recv` and events created using `choose` are composite.

```
composite_event = choose (send c1 m1) (choose (send c2 m2) (send c3 m3))
```

From the API's point of view, composite events resemble a tree with base events in the leaves. However, for the algorithm descriptions here, we consider an event to be a *set* of base events. An implementation could impose an ordering on the base events that make up a composite event. Different orderings correspond to different event-prioritisation algorithms.

In the algorithm descriptions below, a **Channel** consists of two FIFO queues, one for `send` and one for `recv`. These queues store process identities. While blocked on a `recv` on a channel, that process' id is stored in the receive queue of that channel; likewise for `send` and the send-queue. Synchronous exchange means that messages themselves do not need to be maintained on a queue.

Additionally, the algorithms below rely on there being a queue of processes that are ready to execute. This queue is called the `readyQ`. In the algorithm descriptions, handling of `wrap` has been omitted. A function wrapped around an event specifies an operation that should be performed after synchronisation has been completed. Also, we abstract over the synchronisation of hardware events. As a convention, `self` refers to the process from which the `sync` operation is executed.

4.1 Synchronising events

The synchronisation algorithm that performs the API operation `sync` accepts a set of base events. It searches the set of events for a base event that has a sender or receiver blocked (ready to synchronise) and passes the message between sender and receiver. Algorithm 1 provides a high-level view of the synchronisation algorithm.

The first step in synchronisation is to see if there exists a synchronisable event in the set of base events. The *findSynchronisableEvent* algorithm is presented in Algorithm 2.

If the *findSynchronisableEvent* algorithm is unable to find an event that can be synchronised, the process initiating the synchronisation is blocked. The process identifier then gets added to all the channels involved in the base events of the set. This is shown in Algorithm 3.

After registering the process identifiers on the channels involved, the currently running process should yield its hold on the CPU, allowing another process to run. The next process to start running is found using the *dispatchNewProcess* algorithm in Algorithm 4.

When two processes are communicating, the first one to be scheduled will block as the other participant in the communication is not yet waiting on the channel. However, when *dispatchNewProcess* dispatches the second process, the *findSynchronisableEvent* function will return a synchronisable event and the *syncNow* operation (see Algorithm 5) does the actual message passing.

Algorithm 1 The synchronisation algorithm.

Data: $event : Set$
$ev \leftarrow findSynchronisableEvent(event)$;
if $ev \neq \varnothing$ **then**
  $syncNow(ev)$;
else
  $block(event)$;
  $dispatchNewProcess()$;
end

4.2 Timed synchronisation of events

Timed synchronisation is handled by a two-part algorithm – the first part (Algorithm 6) runs when a process is executing the `syncT` API operation, and the second part (Algorithm 7) is executed later, after the baseline time specified in the `syncT` call is reached.

Algorithm 2 The findSynchronisableEvent function.

```
Data: event : Set
Result: A synchronisable event or ∅
foreach e ∈ event do
    if e.baseEventType == SEND then
        if ¬isEmpty(e.channelNo.recvq) then
            return e
        end
    else if e.baseEventType == RECV then
        if ¬isEmpty(e.channelNo.sendq) then
            return e
        end
    else return ∅;                                   /* Impossible case */
end
return ∅ ;                                 /* No synchronisable event found */
```

Algorithm 3 The block function.

```
Data: event : Set
foreach e ∈ event do
    if e.baseEventType == SEND then
        e.channelNo.sendq.enqueue(self);
    else if e.baseEventType == RECV then
        e.channelNo.recvq.enqueue(self);
    else Do nothing;                                 /* Impossible case */
end
```

Algorithm 4 The dispatchNewProcess function.

```
if readyQ ≠ ∅ then
    process ← dequeue(readyQ);
    currentProcess = process;
else
    relinquish control to the underlying OS
end
```

Algorithm 5 The syncNow function.

```
Data: A base-event value - event
if event.baseEventType == SEND then
    receiver ← dequeue(event.channelNo.recvq);
    deliverMSG(self, receiver, msg) ;      /* pass msg from self to receiver */
    readyQ.enqueue(self);
else if event.baseEventType == RECV then
    sender ← dequeue(event.channelNo.sendq);
    deliverMSG(sender, self, msg) ;          /* pass msg from sender to self */
    readyQ.enqueue(sender);
else Do nothing;                                     /* Impossible case */
```

These algorithms rely on there being an alarm facility based on absolute wall-clock time, which invokes Algorithm 7 at a specific time. The alarm facility provides the operation `setAlarm` used in the algorithms below. The algorithms also require a queue, `waitQ`, to hold processes waiting for their baseline time-point.

Algorithm 6 The time function.

```
Data: Relative Baseline = baseline, Relative Deadline = deadline
T_wakeup = self.T_local + baseline;
if deadline == 0 then
    T_finish = Integer.MAX;                     /* deadline = 0 implies no deadline */
else
    T_finish = T_wakeup + deadline;
end
self.deadline = T_finish;
baseline_absolute = T_absolute + baseline;
deadline_absolute = T_absolute + baseline + deadline;
cond1 = T_absolute > deadline_absolute;
cond2 = (T_absolute ≥ baseline_absolute)&&(T_absolute ≤ deadline_absolute);
cond3 = baseline < ε;                           /* platform dependent small time period */
if baseline == 0 ∨ cond1 ∨ cond2 ∨ cond3 then
    readyQ.enqueue(currentThread);
    dispatchNewProcess();
    return;
end
setAlarm(T_wakeup);
waitQ.enqueue(self).orderBy(T_wakeup);
dispatchNewProcess();
```

The `handleAlarm` function in Algorithm 7 runs when an alarm goes off and, at that point, makes a process from the waitQ ready for execution. When the alarm goes off, the scheduler either preempts the running process or lets it complete using the earliest deadline first policy. In the absence of a running process, the process coming from the waitQ is scheduled.

Algorithm 7 The handleAlarm function.

```
Data: Wakeup Interrupt
timedProcess ← dequeue(waitQ);
T_now = timedProcess.baseline;
timedProcess.T_local = T_now;
if waitQ ≠ ∅ then
    timedProcess2 ← peek(waitQ);                /* Does not dequeue */
    setAlarm(timedProcess2.baseline);
end
if currentProcess == ∅;                         /* No process currently running */
 then
    currentProcess = timedProcess;
else
    if timedProcess.deadline < currentProcess.deadline then
        /* Preempt currently running process                               */
        readyQ.enqueue(currentProcess);
        currentProcess = timedProcess;
    else
        /* Schedule timed process to run after currentProcess              */
        readyQ.enqueue(timedProcess);
        currentProcess.T_local = T_now;         /* Avoids too much time drift */
    end
end
```

5 Implementation in SynchronVM

The algorithms of Section 4 are implemented within the Synchron runtime. The Synchron API and runtime are part of a larger programming platform that is the bytecode-interpreted virtual machine called SynchronVM [2], which builds on the work by Sarkar et al. [26].

The execution unit of SynchronVM is based on the Categorical Abstract Machine (CAM) [7]. CAM supports the cheap creation of closures, as a result of which SynchronVM can support a functional language quite naturally. CAM was chosen primarily for its simplicity and availability of pedagogical resources [15].

5.1 System Overview

The software architecture of SynchronVM consists of three parts – (1) the frontend, (2) the middleware and (3) the backend.

The **frontend** is a statically-typed, eager, Caml-like functional language with Hindley-Milner type inference. Polymorphic types are monomorphised as part of the compilation and there is a lambda-lifting pass to reducce heap-allocation of closures.

In the **middleware** the frontend language is compiled down to an untyped intermediate representation (IR) based on lambda-calculus. The IR is then compiled into bytecode operations for the virtual machine.

The **backend** of the system is the bytecode interpreting virtual machine, currently based on the categorical abstract machine [15] . The VM uses a standard non-moving, mark-and-sweep garbage collector for automated memory management and support for closures. In addition, the VM implements a low-level bridge interface that exposes hardware units to programs as processes that communicate using message passing.

5.1.1 Concurrency, I/O and Timing bytecode instructions

For accessing the operators of our programming interface as well as any general runtime-based operations, SynchronVM has a dedicated bytecode instruction – `CALLRTS n`, where `n` is a natural number to disambiguate between operations. Table 1 shows the bytecode operations corresponding to our programming interface.

Table 1 Concurrency, I/O and Timing bytecodes.

spawn	CALLRTS 0
channel	CALLRTS 1
send	CALLRTS 2

recv	CALLRTS 3
sync	CALLRTS 4
choose	CALLRTS 5

spawnExternal	CALLRTS 6
wrap	CALLRTS 7
syncT	CALLRTS 8; CALLRTS 4

Notably, the `syncT` operation gets compiled into a sequence of two instructions. The first instruction in the `syncT` sequence is `CALLRTS 8` which corresponds to Algorithm 6 in Section 4. When the process is woken up by Algorithm 7, the process program counter lands at the next instruction which is `CALLRTS 4` (`sync`).

5.2 Message-passing with events

All forms of communication and I/O in SynchronVM operate via synchronous message-passing. However, a distinct aspect of SynchronVM's message-passing is the separation between the *intent* of communication and the actual communication. A value of type `Event` indicates the intent to communicate.

An event-value, like a closure, is a concrete runtime value allocated on the heap. The fundamental event-creation primitives are `send` and `recv`, which Reppy calls base-event constructors [25]. The event composition operators like `choose` and `wrap` operate on these base-event values to construct larger events. When a program attempts to send or receive a message, an event-value captures the channel number on which the communication was desired. When this event-value is synchronised (Section 4), we use the channel number as an identifier to match between prospective senders and receivers. Listing 3 shows the heap representation of an event-value as the type `event_t` and the information that the event-value captures on SynchronVM.

Listing 3 Representing an `Event` in SynchronVM.

```
typedef enum {
   SEND , RECV
} event_type_t;

typedef struct {
  event_type_t e_type; //  8 bits
  UUID channel_id;     //  8 bits
} base_evt_simple_t;

typedef struct {
  base_evt_simple_t  evt_details; // stored with 16 bits free
  cam_value_t wrap_func_ptr; // 32 bits
} base_event_t;

typedef struct {
  base_event_t bev; // 32 bits
  cam_value_t  msg; // 32 bits; NULL for recv
} cam_event_t;

typedef heap_index event_t;
```

An event is implemented as a linked list of base-events constructed by applications of the `choose` operation. Each element of the list captures (i) the message that is being sent or received, (ii) any function that is wrapped around the base-event using `wrap`, (iii) the channel being used for communication and (iv) an enum to distinguish whether the base-event is a `send` or `recv`. Fig 3 visualises an event upon allocation to the Synchron runtime's heap.

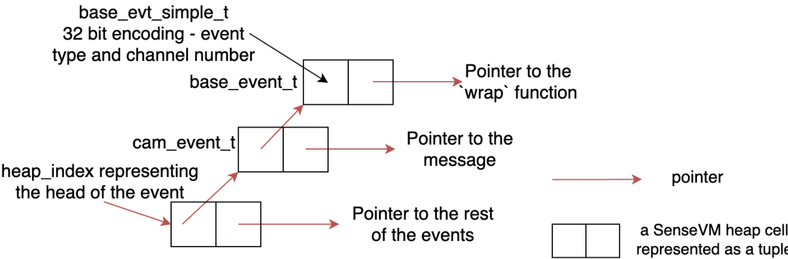


Figure 3 An event on the SynchronVM heap.

The linked-list, as shown above, is the canonical representation of an `Event`-type. It can represent any complex composite event. For instance, if we take an example composite event that is created using the base-events, e_1, e_2, e_3 and a wrapping function wf_1, it can always be rewritten to its canonical form.

```
choose e₁ (wrap (choose e₂ e₃) wf₁)
-- Rewrite to canonical form --
choose e₁ (choose (wrap e₂ wf₁) (wrap e₃ wf₁))
```

The `choose` operation can simply be represented as *cons*ing onto the event list.

5.3 The scheduler

SynchronVM's scheduler is a hyrbid of cooperative and preemptive scheduling. For applications that do not use `syncT`, the scheduler is cooperative in nature. Initially the threads are scheduled in the order that the main method calls them.

```
main = let _ = spawn thread1 in
       let _ = spawn thread2 in
       let _ = spawn thread3 in ...
```

The scheduler orders the above in the order of `thread1` first, `thread2` next and `thread3` last. As the program proceeds, the scheduler relies on the threads to yield their control according to the algorithms of Section 4. When the scheduler is unable to find a matching thread for the currently running thread that is ready to synchronise the communication, it blocks the current thread and calls the $dispatchNewProcess()$ function to run other threads (see Algorithm 1). On the other hand, when synchronisation succeeds, the scheduler puts the message-sending thread in the `readyQ` and the message-receiving thread starts running.

The preemptive behaviour of the scheduler occurs when using `syncT`. For instance, when a particular *untimed* thread is running and the baseline time of a timed thread has arrived, the scheduler then preempts the execution of the *untimed* thread and starts running the timed thread. A similar policy is also observed when the executing thread's deadline is later than a more urgent thread; the thread with the earliest deadline is chosen to be run at that instance. Algorithm 7 shows the preemptive components of the scheduler.

The SynchronVM scheduler also handles hardware driver interactions via message-passing. The structure that is used for messaging is shown below:

Listing 4 A SynchronVM hardware message

```
typedef struct {
  uint32_t sender_id;
  uint32_t msg_type;
  uint32_t data;
  Time timestamp;
} svm_msg_t;
```

The `svm_msg_t` type contains a unique sender id for each driver that is the same as the number used in `spawnExternal` to identify that driver. The 32 bit `msg_type` field can be used to specify different meanings for the next field, the `data`. The `data` is a 32 bit word. The `timestamp` field of a message struct is a 64 bit entity, explained in detail in Section 5.5. When the SynchronVM scheduler has all threads blocked, it uses a function pointer called `blockMsg`, which is passed to it by the OS that starts the scheduler, to wait for any interrupts from the underlying OS (more details in Section 5.4). Upon receiving an interrupt, the scheduler uses the SynchronVM runtime's `handleMsg` function to handle the corresponding message. The function internally takes the message and unblocks the thread for which the message was sent. The general structure of SynchronVM's scheduler is shown in Algorithm 8.

Algorithm 8 The SynchronVM scheduler.

Data: $blockMsg$ function pointer
$\forall threads\ set\ T_{local} = T_{absolute}$;
$svm_msg_t\ msg$;
while *True* **do**
  if *all threads blocked* **then**
    $blockMsg(\&msg)$;
    $handleMsg(msg)$;
  else
    $interpret(currentThread.PC)$;
  end
end

The T_{local} clock is initialised for each thread when starting up the scheduler. Also notable is the `blockMsg` function that relinquishes control to the underlying OS, allowing it to save power. When the interrupt arrives, the `handleMsg` function unblocks certain thread(s) so that when the *if..then* clause ends, in the following iteration the *else* clause is executed and bytecode interpretation continues. We next discuss the low-level bridge connecting the Synchron runtime to the underlying OS.

5.4 The Low-Level Bridge

The low-level bridge specifies two interfaces that should be implemented when writing peripheral drivers to use with SynchronVM. The first contains functions for reading and writing data synchronously to and from a driver. The second is geared towards interrupt-based drivers that asynchronously produce data.

The C-`struct` below contains the interface functions for reading and writing data to a driver as well as functions for checking the availability of data.

```
typedef struct ll_driver_s{
  void *driver_info;
  bool is_synchronous;
  uint32_t (*ll_read_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_write_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_data_readable_fun)(struct ll_driver_s* this);
  uint32_t (*ll_data_writeable_fun)(struct ll_driver_s* this);
  UUID channel_id;
} ll_driver_t;
```

The `driver_info` field in the `ll_driver_t` struct can be used by a driver that implements the interface to keep a pointer to lower-level driver specific data. For interrupt-based drivers, this data will contain, among other things, an *OS interoperation* struct. These OS interoperation structs are shown further below. A boolean indicates whether the driver is synchronous or not. Next, the struct contains function pointers to the low-level driver's implementation of the interface. Lastly, a `channel_id` identifies the channel along which the driver is allowed to communicate with processes running on top of SynchronVM.

The `ll_driver_t` struct contains all the data associated with a driver's configuration in one place and defines a set of platform and driver independent functions for use in the runtime system, shown below:

```
uint32_t ll_read(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_write(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_data_readable(ll_driver_t *drv);
uint32_t ll_data_writeable(ll_driver_t *drv);
```

The OS interoperation structs, mentioned above, are essential for drivers that asynchronously produce data. We show their Zephyr and ChibiOS versions below:

```
typedef struct zephyr_interop_s {
  struct k_msgq *msgq;
  int (*send_message)(struct zephyr_interop_s* this, svm_msg_t msg);
} zephyr_interop_t;

typedef struct chibios_interop_s {
  memory_pool_t *msg_pool;
  mailbox_t *mb;
  int (*send_message)(struct chibios_interop_s* this, svm_msg_t msg);
} chibios_interop_t;
```

In both cases, the struct contains the data that functions need to set up low-level message-passing between the driver and the OS thread running the SynchronVM runtime. Zephyr provides a message-queue abstraction that can take fixed-size messages, while ChibiOS

supports a mailbox abstraction that receives messages that are the size of a pointer. Since ChibiOS mailboxes cannot receive data that is larger than a 32-bit word, a memory pool of messages is employed in that case. The structure used to send messages from the drivers is the already-introduced `svm_msg_t` struct, given in Listing 4.

5.5 The wall-clock time subsystem

Programs running on SynchronVM that make use of the timed operations rely on there being a monotonically increasing timer. The wall-clock time subsystem emulates this by implementing a 64bit timer that would take almost 7000 years to overflow at 84MHz frequency or about 36000 years at 16MHz. The timer frequency of 16MHz is used on the NRF52 board, while the timer runs at 84MHz on the STM32.

SynchronVM requires the implementation of the following functions for each of the platforms (such as ChibiOS and Zephyr) that it runs on:

```
1 bool      sys_time_init(void *os_interop);
2 Time      sys_time_get_current_ticks(void);
3 uint32_t  sys_time_get_alarm_channels(void);
4 uint32_t  sys_time_get_clock_freq(void);
5 bool      sys_time_set_wake_up(Time absolute);
6 Time      sys_time_get_wake_up_time(void);
7 bool      sys_time_is_alarm_set(void);
```

The timing subsystem uses the same OS interoperation structs as drivers do and thus has access to a communication channel to the SynchronVM scheduler. The interoperation is provided to the subsystem at initialisation using `sys_time_init`.

The key functionality of the timing subsystem is the ability to set an alarm at an absolute 64-bit point in time. Setting an alarm is done using `sys_time_set_wake_up`. The runtime system queries the timing subsystem to check if an alarm is set and at what specific time.

The low-level implementation of the timing subsystem is highly platform dependent at present. But on both Zephyr and ChibiOS, the implementation is currently based on a single 32-bit counter configured to issue interrupts at overflow, where an additional 32-bit value is incremented. Alarms can only be set on the lower 32-bit counter at absolute 32-bit values. Additional logic is needed to translate between the 64-bit alarms set by SynchronVM and the 32-bit timers of the target platforms. Each time the overflow interrupt happens, the interrupt service routine checks if there is an alarm in the next 32-bit window of time and in that case, enables a compare interrupt to handle that alarm. When the alarm interrupt happens, a message is sent to the SynchronVM scheduler in the same way as for interrupt based drivers, using the message queue or mailbox from the OS interoperation structure.

Revisiting the requirements for implementing `syncT` (Section 3.3), we find that our scheduler (1) provides a preemptive mechanism to override the fair scheduling, (2) has access to a wall-clock time source, and (3) implements an earliest-deadline-first scheduling policy that attempts to match the local time and the absolute time.

5.6 Porting SynchronVM to another RTOS

For porting SynchronVM to a new RTOS, one needs to implement – (1) the wall-clock time subsystem interface from Section 5.5, (2) the low-level bridge interface (Section 5.4) for each peripheral, and (3) a mailbox or message queue for communication between asynchronous drivers and the runtime system, required by the time subsystem.

Our initial platform of choice was ZephyrOS for its platform-independent abstractions. The first port of SynchronVM was on ChibiOS, where the wall-clock time subsystem was 254 lines of C-code. The drivers for LED, PWM, and DAC were about 100 lines of C-code each.

6 Case Studies

Finite-State Machines with Synchron

We will begin with two examples of expressing state machines (involving callbacks) in the Synchron API. Our examples are run on the NRF52840DK microcontroller board containing four buttons and four LEDs. We particularly choose the button peripheral because its drivers have a callback-based API that typically leads to non-linear control-flows in programs.

6.1 Four-Button-Blinky

We extend the *button-blinky* example (see Listing 1) to produce a one-to-one mapping between four LEDs and four buttons such that button1 press lights up LED1, button2 lights up LED2, button3 lights up LED3 and button4 lights up LED4 (while the button releases switch off the corresponding LEDs).

The state machine of button-blinky is a standard two-state automaton that moves from the ON-state to OFF on button-press and vice versa. Now, for the four button-LED combinations, we have four state machines. We can combine them using the `choose` operator.

Listing 5 shows the important parts of the logic. The four state machines are declared in Lines 1 to 4, and their composition happens in Line 6 using the `choose` operator.

Listing 5 The Four-Button-Blinky program expressed in the Synchron API.

```
press1 = wrap (recv butchan1) (λ x -> sync (send ledchan1 x))
press2 = wrap (recv butchan2) (λ x -> sync (send ledchan2 x))
press3 = wrap (recv butchan3) (λ x -> sync (send ledchan3 x))
press4 = wrap (recv butchan4) (λ x -> sync (send ledchan4 x))

anybutton = choose press1 (choose press2 (choose press3 press4))

program : ()
program = let _ = sync anybutton in program
```

6.2 A more intricate FSM

We now construct a more intricate finite-state machine involving intermediate states that can move to an error state if the desired state-transition buttons are not pressed. For this example a button driver needs to be configured to send only one message per button press-and-release. So there is no separate button-on and button-off signal but one signal per button.

In this FSM, we glow the LED1 upon consecutive presses of button1 and button2. We use the same path to turn LED1 off. However, if a press on button1 is followed by a press of button 1 or 3 or 4, then we move to an error state indicated by LED3. We use the same path to switch off LED3. In a similar vein, consecutive presses of button3 and button4 turns on LED2 and button3 followed by button 1 or 2 or 3 turns on the error LED – LED3. Fig. 4 shows the FSM diagram of this application, omitting self-loops in the OFF state.

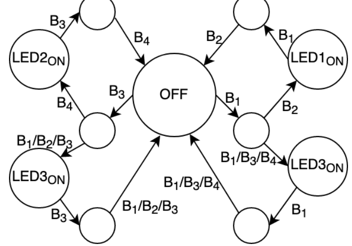


Figure 4 A complex state machine.

Listing 6 shows the central logic expressing the FSM of Fig 4 in the Synchron API. This FSM can be viewed as a composition of two separate finite state machines, one on the left side of the OFF state involving LED2 and LED3 and one on the right side involving LED1 and LED3. Once again, we use the `choose` operator to compose these two state machines.

Listing 6 The complex state machine running on the SynchronVM.

```
errorLed x = ledchan3

fail1ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan3) errorLed)
                 (wrap (recv butchan4) errorLed))

fail2ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan2) errorLed)
                 (wrap (recv butchan3) errorLed))

led1Handler x =
   sync (choose (wrap (recv butchan2) (\x -> ledchan1)) fail1ev)

led2Handler x =
   sync (choose (wrap (recv butchan4) (\x -> ledchan2)) fail2ev)

led : Int -> ()
led state =
  let fsm1 = wrap (recv butchan1) led1Handler in
  let fsm2 = wrap (recv butchan3) led2Handler in
  let ch = sync (choose fsm1 fsm2) in
  let _ = sync (send ch (not state)) in
  led (not state)
```

In Listing 6, the `led1Handler1` and `ledHandler2` functions capture the intermediate states after one button press, when the program awaits the next button press. The error states are composed using the `choose` operator in the functions `fail1ev` and `fail2ev`.

The compositional nature of our framework is visible in line no. 21 where we compose the two state machines, `fsm1` and `fsm2`, using the `choose` operator. Synchronising on this composite event returns the LED channel (demonstrating a higher-order approach) on which the process should attempt to write. This program is notably a highly callback-based, reactive program that we have managed to represent in an entirely synchronous framework.

6.3 A soft-realtime music playing example

We present a soft-realtime music playing exercise from a Real-Time Systems course, expressed using the Synchron API. We choose the popular nursery rhyme – “Twinkle, Twinkle, Little Star”. The program plays the tune repeatedly until it is stopped.

The core logic of the program involves periodically writing a sequence of 1’s and 0’s to a DAC driver. However, to make the produced note sound musical to the human ear, the *periodic rate* at which our process writes to the DAC driver is important, and this is where the real-time aspect of the application comes in. The human ear recognises a note produced at a certain frequency as a musical note. Our sound is generated at the 196Hz G3 music key.

Listing 7 shows the principal logic of the program expressed using the Synchron API. Note that we use `syncT` to describe a new temporal combinator `after` that determines the periodicity of this program. The list `twinkle` (line 10) holds the 28 notes in the twinkle song and the list `durations` (line 11) provides the length of each note.

Listing 7 The *Twinkle, Twinkle* tune expressed using the Synchron API.

```
msec t = t * 1000
usec t = t
after t ev = syncT t 0 ev
-- note frequencies
g = usec 2551 -- a = 2273 usecs , b = 2025 usecs and so on

hn = msec 1000 -- half note
qn = msec  500 -- quarter note

twinkle   = [ g,  g,  d,  d,  e,  e,  d.... ] -- 28 notes
durations = [qn, qn, qn, qn, qn, qn, hn.... ]

dacC  = channel ()
noteC = channel ()

playerP : List Int -> List Int -> Int -> () -> ()
playerP melody nt n void =
  if (n == 29)
  then let _ = after (head nt) (send noteC (head twinkle)) in
       playerP (tail twinkle) durations 2 void
  else let _ = after (head nt) (send noteC (head melody)) in
       playerP (tail melody) (tail nt) (n + 1) void

tuneP : Int -> Int -> () -> ()
tuneP timePeriod vol void =
  let newtp =
      after timePeriod (choose (recv noteC)
                               (wrap (send dacC (vol * 4095))
                                     (λ _ -> timePeriod))) in
  tuneP newtp (not vol) void

main = let _ = spawnExternal dacC 0 in
       let _ = spawn (tuneP (head twinkle) 1) in
       let _ = spawn (playerP (tail twinkle) durations 2) in ()
```

The application consists of two software processes and one external hardware process. We use two channels – `dacC` to communicate with the DAC and `noteC` to communicate between the two software processes. Looking at what each software process is doing –

playerP. This process runs at the rate of a note's length. For a quarter note it wakes up after 500 milliseconds (1000 msecs for a half note), traverses the next element of the `twinkle` list and sends it along the `noteC` channel. It circles back after completing all 28 notes.

tuneP. This process creates the actual sound. Its running rate varies depending on the note that is being played. For instance, when playing note C, it will write to the DAC at a rate of 1911 microseconds-per-write. However, upon receiving a new value along `noteC`, it changes its write frequency to the new value resulting in changing the note of the song.

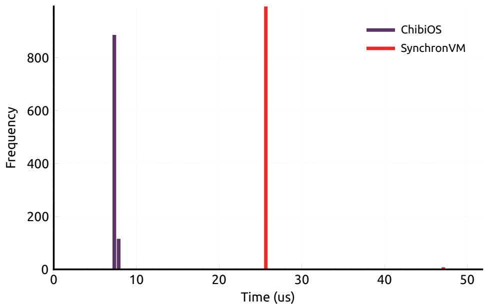
7 Benchmarks

7.1 Interpretive overhead measurements

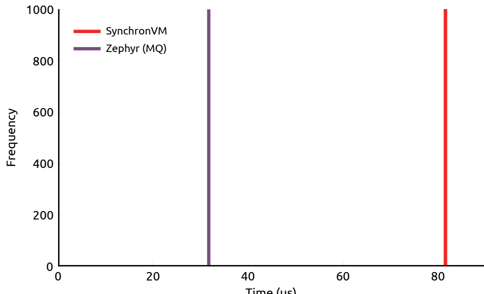
We characterise the overhead of executing programs on top of SynchronVM, compared to running them directly on either Zephyr or ChibiOS, by implementing *button-blinky* directly on top of these operating systems and measuring the response-time differences.

We compare the interrupt-based button-blinky implementation of Zephyr and ChibiOS with the corresponding SynchronVM program. The interrupt-based approach (as opposed to polling) keeps the low-level implementation in Zephyr and ChibiOS similar to SynchronVM and indicates the interpretive and other overheads in SynchronVM.

The data in charts presented here is collected using an STM32F4 microcontroller based testing system connected to either the NRF52 or the STM32F4 system under test (SUT). The testing system provides the stimuli, sets the GPIO (button) to either active or inactive and measures the time it takes for the SUT to respond on another GPIO pin (symbolising the LED). The testing system connects to a computer displaying a GUI and generates the plots used in this paper. Each plot places measured response times into buckets of similar time, and shows the number of samples falling in each bucket as a vertical bar. Each bucket is labelled with the average time of the samples it contains.



(a) Response time comparison between a C-code implementation using ChibiOS against the same program on SynchronVM (running on ChibiOS). Data obtained on the STM32F4 microcontroller. Uses 1000 samples.



(b) Response time comparison between a C-code implementation using Zephyr OS against the same program on SynchronVM (running on Zephyr). Data obtained on the NRF52 microcontroller. Uses 1000 samples.

Figure 5 Button-blinky response times comparison between C and SynchronVM.

Fig. 5a shows the SynchronVM response time in comparison to the implementation of the program running on ChibiOS using its mailbox abstraction (MB). There the overhead is about 3x. Fig. 5b compares response times for SynchronVM and the Zephyr message queue based implementation (MQ), and shows an overhead of 2.6x.

7.2 Effects of Garbage Collection

This experiment measures the effects of garbage collection on response time by repeatedly running 10000 samples for different heap-size configurations of SynchronVM. A smaller heap should lead to more frequent interactions with the garbage collector, and the effects of the garbage collector on the response time should magnify.

As a smaller heap is used, the number of outliers should increase if the outliers are due to garbage collection. The following table shows the number of outliers at each size configuration for the heap used, and there is an indication that GC is the cause of outliers.

Heap size (bytes)	256	512	1024	2048	4096	8192
Outliers NRF52 on Zephyr	3334	1429	811	491	0	81
Outliers STM32 on ChibiOs	3339	1430	810	491	0	80

Figures 6 and 7 show the response-time numbers across the heap sizes of 8192, 4096, 2048, 1024, 512 and 256 bytes. A general observable trend is that as the heap size decreases and GC increases, the response time numbers hover towards the farther end of the X-axis. This

trend is most visible for the heap size of 256 bytes, which is our smallest heap size. Note that we cannot collect enough sample data for response-time if we switch off the garbage collector (as a reference value), as the program would very quickly run out of memory and terminate.

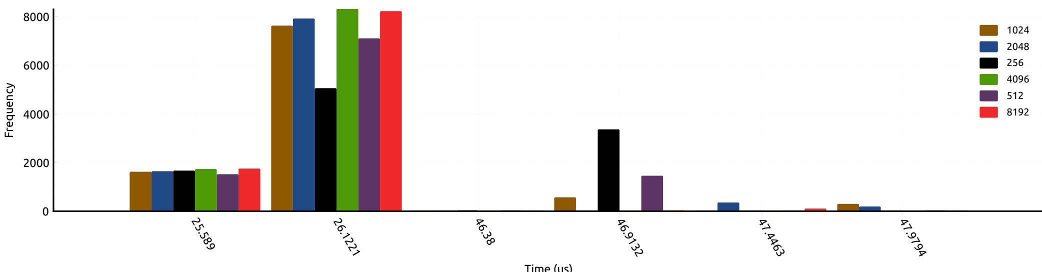


Figure 6 Response time measurements at different sizes of the heap to identify effects of garbage collection. This data is collected on the STM32F4 microcontroller running SynchronVM on top of ChibiOS. Each bucket size is approx 0.533us. Uses 10000 samples.

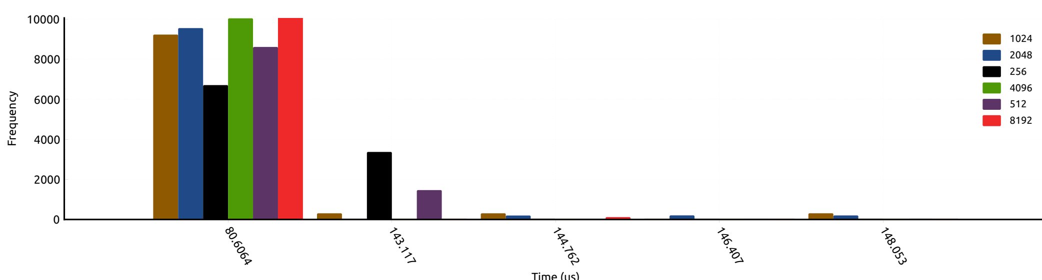


Figure 7 Response time measurements at different sizes of the heap to identify effects of garbage collection. This data is collected on the NRF52 microcontroller running SynchronVM on top of the Zephyr OS. Each bucket size is approx 1.65us. Uses 10000 samples.

7.3 Memory Footprint

SynchronVM resides on the Flash memory of a microcontroller. On Zephyr, a tiny C application occupies 17100 bytes, whereas the same SynchronVM application occupies 49356 bytes, which gives the VM's footprint as 32256 bytes. For ChibiOS, the C application takes 18548 bytes, while the SynchronVM application takes 53868 bytes. Thus, SynchronVM takes 35320 bytes in this case. Hence, we can estimate SynchronVM's rough memory footprint at 32 KB, which will grow with more drivers.

7.4 Power Usage

Fig. 8 shows the power usage of the NRF52 microcontroller running the button-blinky program for three implementations. The first is a polling version of the program in C. The second program uses a callback-based version of button-blinky [11]. The last program is Listing 1 running on SynchronVM. The measurements are made using the Ruideng UM25C ammeter. We collect momentary readings from the ammeter after the value has stabilised.

Notable in Fig. 8 is the polling-based C implementation's use of 0.0175 Watts of power in a button-off state, whereas SynchronVM consumes five times less power (0.0035 Watts).

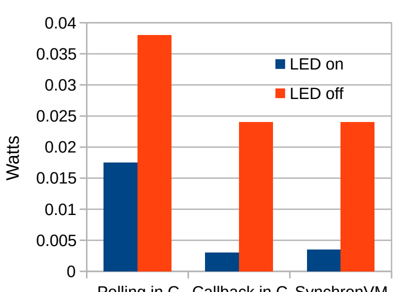


Figure 8 Power usage measured on the NRF52 microcontroller.

This is comparable to the callback-based C implementation's use of 0.003 Watts. Integrating the power usage over time will likely make the difference between SynchronVM and the callback-based C version more noticeable. However, we believe that the simplicity and declarative nature of the Synchron-based code provide a fair tradeoff.

7.5 Jitter and Precision

Jitter can be defined as the deviation from true periodicity of a presumably periodic signal, often in relation to a reference clock signal. We want to evaluate how our claims of `syncT` reducing jitter pans out in practice.

Listing 8 below is written in a naive way to illustrate how jitter manifests in programs. Figure 9a shows what the oscilloscope draws, set to persistent mode drawing while sampling the signal from the Raspberry Pi outputs.

The Raspberry Pi program reads the status of a GPIO pin and then inverts its state back to that same pin. The program then goes to sleep using `usleep` for 400us. The goal frequency was 1kHz and sleeping for 400us here gave a roughly 1.05kHz signal. The more expected sleep time of 500us to generate a 1kHz signal led, instead, to a much lower frequency. So, the 400us value was found experimentally.

```
1  while (1) {
2   uint32_t state = GPIO_READ(23);
3   if (state) {
4     GPIO_CLR(23);
5   } else {
6     GPIO_SET(23);
7   }
8   usleep(400);
9   }
10 // main method and other setup
       elided
```

Listing 8 Raspberry Pi C code.

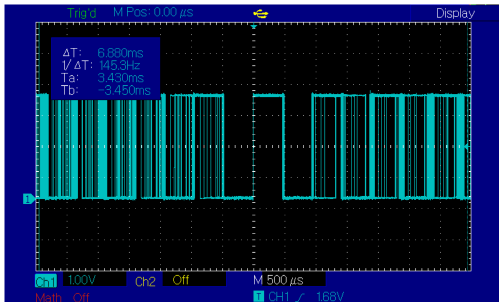
```
11 ledchan = channel ()
12
13 foo : Int -> ()
14 foo val =
15  let _ = syncT 500 0 (send
        ledchan val)
16   in foo (not val)
17
18 main =
19  let _ = spawnExternal ledchan 1
20   in foo 1
```

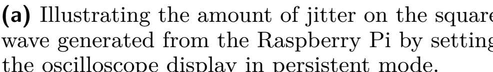
Listing 9 SynchronVM 1KHz wave code.

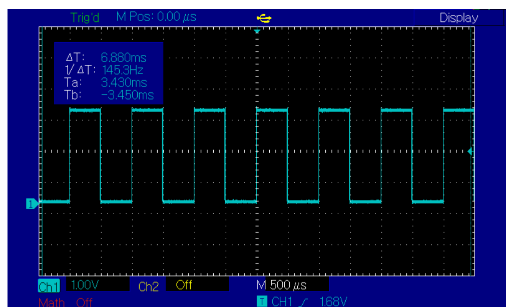
Listing 9 shows the same 1kHz frequency generator for SynchronVM. Note that, in this case, specifying a baseline of 500us led to a 1kHz wave (Fig. 9b). In comparison, a 400us period in Listing 8 generated a roughly 1kHz wave, owing to additional delays of the system.

7.6 Load Test

The SynchronVM program in the previous section could produce a 1kHz-wave with no jitter. However, the only operation that the program did was produce the square wave. In this section, we want to test how much computational load can be performed by Synchron while producing the square wave. We emulate the workload using the following program.




(a) Illustrating the amount of jitter on the square wave generated from the Raspberry Pi by setting the oscilloscope display in persistent mode.



(b) A 1kHz square wave generated using SynchronVM running on the STM32F4 with no jitter.

Figure 9 A 1 kHz frequency generator on the Raspberry PI (in C) and STM32 (Synchron).

```
21 load i n =
22   let _ = fib_tailrec n in
23   let _ = syncT 8000 0 (send
         ledchan i)
24    in load (not i)
```

```
25 loop i a b n =
26   if i == n then a
27   else loop (i+1) (b) (a+b) n
28
29 fib_tailrec n = loop 0 0 1 n
```

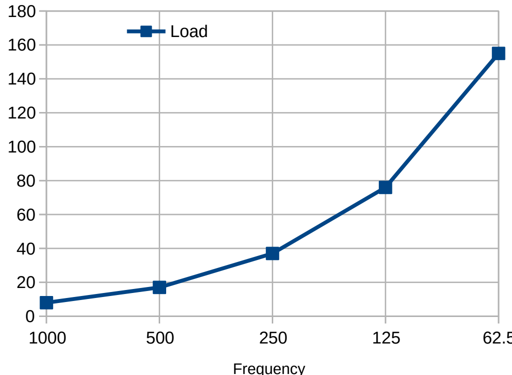


Figure 10 Load testing SynchronVM with the nth fibonacci number function.

At a given frequency, it is possible to calculate only up to a certain Fibonacci number while generating the square wave at the desired frequency. For example, when generating a 62.5 Hz wave, it is only possible to calculate up to the 155th Fibonacci number. If the 156th number is calculated, the wave frequency drops below 62.5 Hz.

Fig. 10 plots the nth Fibonacci numbers that can be calculated against the square wave frequencies that get generated without jitters. Our implementation of `fib_tailrec` involves 2 addition operations, 1 equality comparison and 1 recursive call. So, calculating the 155th Fibonacci number involves $155 * 4 = 620$ major operations. The trend shows that the load capacity of SynchronVM grows linearly as the desired frequency of the square wave is halved.

7.7 Music Program Benchmarks

We now provide some benchmarks on the music program from Section 6.3. Figure 11 shows CPU usage, average time it takes to allocate data and total time spent doing allocations in a 1 minute window. The values used in the chart come from the second minute of running

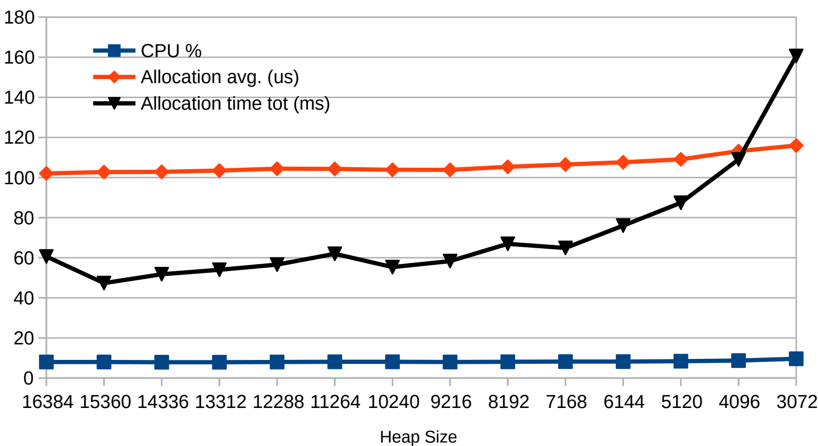


Figure 11 CPU usage and allocation trends over a 1 minute window for Listing 7.

the music application. The values from the first minute of execution are discarded as those would include the startup phase of the system. The amount of heap made available to the runtime system is varied from a roomy 16384 bytes down to 3072 bytes.

The sweep phase of our garbage collector is intertwined with the allocations phase. Hence, instead of showing the GC time, the chart shows statistics related to all allocations that take a measurable amount of time using the ChibiOS 10KHz system timer. All allocations taking less than 100us are left out of the statistics (and not counted towards averaging).

The data in Fig. 11 shows that CPU usage of the music application is pretty stable at around 8 percent over the one minute window. It increases slightly for the very small heap sizes and ends up at nearly 10 percent at the smallest heap size that can house the program.

In terms of allocation, the average time of an allocation (in usecs) increases when the probability of a more expensive allocation increases, which in turn increases with small heap sizes. In the last data series, the total amount of time spent in allocations (in msecs) grows considerably as the heap size drops below 7168 bytes – an indicator of increased GC activity.

Programming Complexity. This music application ports a Real-Time Systems course exercise written in C using the TinyTimber kernel [19]. The TinyTimber-based C program (excluding the kernel) is around 600 lines of code. In comparison, our entire program, along with the library functions, stands at 74 lines. The most time-consuming part of the port was modifying the core logic in terms of message-passing. A major gain is that the interrupt-handling and other I/O management routines are invisible to the programmer. The user-defined timing operator, `after`, further enables the concision of the program.

7.8 Discussion

Our benchmarks, so far, show promising results for power, memory, and CPU usage. However, SynchronVM's response time is 2-3x times slower than native C code, which needs improvement. We attribute the slowness to the CAM-based execution engine, which we hope to mitigate by moving to a ZAM-based machine [18].

Our synthetic load test (Fig. 10) indicates that the VM can support around 150 operations for applications that operate around 250Hz (such as humanoid balance bots [9], autonomous vehicle platforms [34]). Our music program falls in the range of 200-500 Hz, and SynchronVM

could sustain that frequency without introducing any jitter. There exist other *untimed*, aperiodic applications with much lower frequencies where SynchronVM could be applicable. Examples include smart home applications [32], monitoring systems [13], etc.

The Synchron API chooses a synchronous message-passing model, unlike actor-based systems, like Erlang, that support an asynchronous message-passing model with each process containing a mailbox. We believe that a synchronous message-passing policy is better suited for embedded systems for the following reasons:

1. Embedded systems are highly memory-constrained, and asynchronous send semantics assume the *unboundedness* of an actor's mailbox, which is a poor assumption in the presence of memory constraints. Once the mailbox becomes full, message-sending becomes blocking, which is already the default semantics of synchronous message-passing.
2. Acknowledgement is implicit in synchronous message-passing systems, in contrast to explicit message acknowledgement in asynchronous systems that leads to code bloat. Additionally, forgetting to remove acknowledgement messages from an actor's mailbox can lead to memory leaks.

8 Limitations and Future Work

In this section, we propose future work to improve the Synchron API and runtime.

8.1 Synchron API limitation

Deadline miss API. Currently, the Synchron API cannot represent actions that should happen if a task were to miss its deadline. We envision adapting the negative acknowledgement API of CML to represent missed-deadline handlers for Synchron.

8.2 SynchronVM limitations

Memory management. A primary area of improvement is upgrading our stop-the-world mark and sweep garbage collector and investigating real-time garbage collectors like Schism [23]. Another relevant future work would be investigating static memory-management schemes like regions [30] and techniques combining regions with GC [14].

Interpretation overhead. A possible approach to reducing our interpretation overhead could be pre-compiling our bytecode to machine code (AOT compilation). Similarly, dynamic optimization approaches like JITing could be an area of investigation.

Priority inversions. Although TinyTimber-style dynamic priorities might reduce priority inversion occurrences, they can still occur on the SynchronVM. Advanced approaches like priority inheritance protocols [27] need to be experimented with on our scheduler.

9 Related Work

Among functional languages running on microcontrollers, there exists OCaml running on OMicroB [33], Scheme running on Picobit [29] and Erlang running on AtomVM [5]. Synchron differs from these projects in the aspect that we identify certain fundamental characteristics of embedded systems and accordingly design an API and runtime to address those demands. As a result, our programming interface aligns more naturally to the requirements of an embedded systems application, in contrast with general-purpose languages like Scheme.

The Medusa [3] language and runtime is the inspiration behind our uniform framework of concurrency and I/O. Medusa, however, does not provide any timing based APIs, and their message-passing framework is based on the actor model (See Section 7.8).

In the real-time space, a safety-critical VM that can provide hard real-time guarantees on Real-Time Java programs is the FijiVM [24] implementation. A critical innovation of the project was the Schism real-time garbage collector [23], from which we hope to draw inspiration for future work on memory management.

RTMLton [28] is another example of a real-time project supporting a general-purpose language like SML. RTMLton adapts the MLton runtime [36] with ideas from FijiVM to enable handling real-time constraints in SML. CML is available as an SML library, so RTMLton provides access to the event framework of CML but lacks the uniform concurrency-I/O model and the `syncT` operator of Synchron.

The Timber language [6] is an object-oriented language that inspired the `syncT` API of Synchron. Timber was designed for hard real-time scenarios; related work on estimating heap space bounds [17] could perhaps benefit our future research.

The WebAssembly project (WASM) has spawned sub-projects like WebAssembly Micro Runtime (WAMR) [1] so that languages that compile to WASM can run on microcontrollers. Notable here is that while several general-purpose languages, like JavaScript, can execute on ARM architectures by compiling to WebAssembly, they lack the native support for the concurrent, I/O-bound, and timing-aware programs that is naturally provided by our API and its implementation. Reactive extensions of Javascript, like HipHop.js [4], are being envisioned to be used for embedded systems.

Another related line of work is embedding domain-specific languages like Ivory [10] and Copilot [22] in Haskell to generate C programs that can run on embedded devices. This approach differs from ours in the aspect that two separate languages dictate the programming model of an EDSL – the first being the DSL itself and the second being the host language (Haskell). We assess that having a single language (like in Synchron) provides a more uniform programming model to the programmer. However, code-generating EDSLs have very little runtime overheads and, when fully optimised, can produce high performance C.

10 Conclusion

In this paper, we have presented Synchron – an API and runtime for embedded systems, which we implement within the larger SynchronVM. We identified three essential characteristics of embedded applications, namely being concurrent, I/O–bound, and timing-aware, and correspondingly designed our API to address all three concerns. Our evaluations, conducted on the STM32 and NRF52 microcontrollers, show encouraging results for power, memory and CPU usage of the SynchronVM. Our response time numbers are within the range of 2-3x times that of native C programs, which we envision being improved by moving to a register-based execution engine and by using smarter memory-management strategies. We have additionally demonstrated the expressivity of our API through state machine-based examples, commonly found in embedded systems. Finally, we illustrated our timing API by expressing a soft real-time application, and we expect further theoretical investigations on the worst-case execution time and schedulability analysis on SynchronVM.

References

1 WAMR – WebAssembly Micro Runtime, 2019. URL: https://github.com/bytecodealliance/wasm-micro-runtime.

2 Synchron Virtual Machine, 2022. URL: https://github.com/SynchronVM/SynchronVM.

3. Thomas W. Barr and Scott Rixner. Medusa: Managing Concurrency and Communication in Embedded Systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 439–450. USENIX Association, 2014. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/barr`.
4. Gérard Berry and Manuel Serrano. Hiphop.js: (A)Synchronous reactive web programming. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 533–545. ACM, 2020. `doi:10.1145/3385412.3385984`.
5. Davide Bettio. AtomVM, 2017. URL: `https://github.com/bettio/AtomVM`.
6. Andrew P Black, Magnus Carlsson, Mark P Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical report, OGI School of Science and Engineering, Oregon Health and Sciences University, Technical Report CSE 02-002. April 2002, 2002.
7. Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 1985. `doi:10.1007/3-540-15975-4_29`.
8. Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. The Synchronous Hypothesis and Synchronous Languages. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. `doi:10.1201/9781420038163.ch8`.
9. Ahmed Elhasairi and Alexandre N. Pechev. Humanoid Robot Balance Control Using the Spherical Inverted Pendulum Mode. *Frontiers Robotics AI*, 2:21, 2015. `doi:10.3389/frobt.2015.00021`.
10. Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. Guilt free ivory. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 189–200. ACM, 2015. `doi:10.1145/2804302.2804318`.
11. Zephyr examples. Zephyr button blinky, 2021. URL: `https://pastecode.io/s/szpf673u`.
12. Damien George. Micropython, 2014. URL: `https://micropython.org/`.
13. R. Kingsy Grace and S. Manju. A Comprehensive Review of Wireless Sensor Networks Based Air Pollution Monitoring Systems. *Wirel. Pers. Commun.*, 108(4):2499–2515, 2019. `doi:10.1007/s11277-019-06535-3`.
14. Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining Region Inference and Garbage Collection. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 141–152. ACM, 2002. `doi:10.1145/512529.512547`.
15. Ralf Hinze. The Categorical Abstract Machine: Basics and Enhancements. Technical report, University of Bonn, 1993.
16. C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978. `doi:10.1145/359576.359585`.
17. Martin Kero, Pawel Pietrzak, and Johan Nordlander. Live Heap Space Bounds for Real-Time Systems. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010. `doi:10.1007/978-3-642-17164-2_20`.
18. Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. PhD thesis, INRIA, 1990.
19. Per Lindgren, Johan Eriksson, Simon Aittamaa, and Johan Nordlander. TinyTimber, Reactive Objects in C for Real-Time Embedded Systems. In *2008 Design, Automation and Test in Europe*, pages 1382–1385, 2008. `doi:10.1109/DATE.2008.4484933`.

20 Tommi Mikkonen and Antero Taivalsaari. Web Applications - Spaghetti Code for the 21st Century. In Walter Dosch, Roger Y. Lee, Petr Tuma, and Thierry Coupaye, editors, *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*, pages 319–328. IEEE Computer Society, 2008. doi:10.1109/SERA.2008.16.

21 Johan Nordlander. *Programming with the TinyTimber kernel*. Luleå tekniska universitet, 2007.

22 Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A Hard Real-Time Runtime Monitor. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2010. doi:10.1007/978-3-642-16612-9_26.

23 Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 146–159. ACM, 2010. doi:10.1145/1806596.1806615.

24 Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time Java on resource-constrained platforms with Fiji VM. In M. Teresa Higuera-Toledano and Martin Schoeberl, editors, *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*, ACM International Conference Proceeding Series, pages 110–119. ACM, 2009. doi:10.1145/1620405.1620421.

25 John H. Reppy. Concurrent ML: Design, Application and Semantics. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1993. doi:10.1007/3-540-56883-2_10.

26 Abhiroop Sarkar, Robert Krook, Bo Joel Svensson, and Mary Sheeran. Higher-Order Concurrency for Microcontrollers. In Herbert Kuchen and Jeremy Singer, editors, *MPLR '21: 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Münster, Germany, September 29-30, 2021*, pages 26–35. ACM, 2021. doi:10.1145/3475738.3480716.

27 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

28 Bhargav Shivkumar, Jeffrey C. Murphy, and Lukasz Ziarek. RTMLton: An SML Runtime for Real-Time Systems. In Ekaterina Komendantskaya and Yanhong Annie Liu, editors, *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings*, volume 12007 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2020. doi:10.1007/978-3-030-39197-3_8.

29 Vincent St-Amour and Marc Feeley. PICOBIT: A Compact Scheme System for Microcontrollers. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. doi:10.1007/978-3-642-16478-1_1.

30 Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Inf. Comput.*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

31 Hideyuki Tokuda, Clifford W. Mercer, Yutaka Ishikawa, and Thomas E. Marchok. Priority Inversions in Real-Time Communication. In *Proceedings of the Real-Time Systems Symposium - 1989, Santa Monica, California, USA, December 1989*, pages 348–359. IEEE Computer Society, 1989. doi:10.1109/REAL.1989.63587.

32 Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical Trigger-Action Programming in the Smart Home. In Matt Jones, Philippe A. Palanque, Albrecht Schmidt, and Tovi Grossman, editors, *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, pages 803–812. ACM, 2014. doi:10.1145/2556288.2557420.

33 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

34 Benjamin Vedder, Jonny Vinter, and Magnus Jonsson. A Low-Cost Model Vehicle Testbed with Accurate Positioning for Autonomous Driving. *J. Robotics*, 2018:4907536:1–4907536:10, 2018. doi:10.1155/2018/4907536.

35 Ge Wang and Perry R. Cook. ChucK: A Concurrent, On-the-fly, Audio Programming Language. In *Proceedings of the 2003 International Computer Music Conference, ICMC 2003, Singapore, September 29 - October 4, 2003*. Michigan Publishing, 2003. URL: http://hdl.handle.net/2027/spo.bbp2372.2003.055.

36 Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, page 1. ACM, 2006. doi:10.1145/1159876.1159877.

37 Gordon Williams. Espruino, 2012. URL: http://www.espruino.com/.

Direct Foundations for Compositional Programming

Andong Fan[1]
Zhejiang University, Hangzhou, China

Xuejing Huang[1]
The University of Hong Kong, China

Han Xu
Peking University, Beijing, China

Yaozhu Sun
The University of Hong Kong, China

Bruno C. d. S. Oliveira
The University of Hong Kong, China

Abstract

The recently proposed CP language adopts Compositional Programming: a new modular programming style that solves challenging problems such as the Expression Problem. CP is implemented on top of a polymorphic core language with disjoint intersection types called F_i^+. The semantics of F_i^+ employs an elaboration to a target language and relies on a sophisticated proof technique to prove the *coherence* of the elaboration. Unfortunately, the proof technique is technically challenging and hard to scale to many common features, including recursion or impredicative polymorphism. Thus, the original formulation of F_i^+ does not support the two later features, which creates a gap between theory and practice, since CP fundamentally relies on them.

This paper presents a new formulation of F_i^+ based on a *type-directed operational semantics* (TDOS). The TDOS approach was recently proposed to model the semantics of languages with disjoint intersection types (but without polymorphism). Our work shows that the TDOS approach can be extended to languages with disjoint polymorphism and model the full F_i^+ calculus. Unlike the elaboration semantics, which gives the semantics to F_i^+ indirectly via a target language, the TDOS approach gives a semantics to F_i^+ directly. With a TDOS, there is no need for a coherence proof. Instead, we can simply prove that the semantics is *deterministic*. The proof of determinism only uses simple reasoning techniques, such as straightforward induction, and is able to handle problematic features such as recursion and impredicative polymorphism. This removes the gap between theory and practice and validates the original proofs of correctness for CP. We formalized the TDOS variant of the F_i^+ calculus and all its proofs in the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Intersection types, disjoint polymorphism, operational semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.18

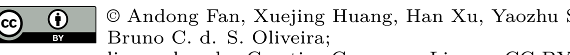
Related Version *Extended Version*: `https://arxiv.org/abs/2205.06150`

Supplementary Material Supplements can be found as follows:
Software (ECOOP 2022 approved artifact): `https://doi.org/10.4230/DARTS.8.2.4`
Software (Coq formalization): `https://github.com/andongfan/CP-Foundations`
Software (Online demo of CP implementation): `https://plground.org`

Funding This research was funded by the University of Hong Kong and Hong Kong Research Grants Council projects number 17209519, 17209520 and 17209821.

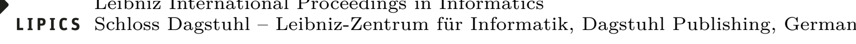
Acknowledgements We thank the anonymous reviewers and Wenjia Ye for their helpful comments.

[1] The first two authors contributed equally to this work.



© Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 18; pp. 18:1–18:28



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Compositional Programming [46] is a recently proposed modular programming paradigm. It offers a natural solution to the *Expression Problem* [42] and novel approaches to modular pattern matching and dependency injection. The CP language adopts Compositional Programming. In CP, several new programming language constructs enable Compositional Programming. Of particular interest for this paper, CP has a notion of *typed first-class traits* [5], which are extended in CP to also enable a form of *family polymorphism* [16].

The semantics of CP and its notion of traits is defined via an elaboration to the core calculus F_i^+ [7]: a polymorphic core language with a *merge operator* [34] and *disjoint intersection types* [30]. The elaboration of traits is inspired by Cook's *denotational semantics of inheritance* [12]. In the denotational semantics of inheritance, the key idea is that mechanisms such as classes or traits, which support *self-references* (a.k.a. the `this` keyword in conventional OOP languages), can be modeled via *open recursion*. In other words, the encoding of classes or traits is parametrized by a self-reference. This allows late binding of self-references at the point of instantiation and enables the modification and composition of traits before instantiation. Instantiation happens when **`new`** is used, just as in conventional OOP languages. When **`new`** is used, it essentially closes the recursion by binding the self-reference, which then becomes a recursive reference to the instantiated object. In the denotational semantics of inheritance, **`new`** is just a fixpoint operator.

The semantics of the original formulation of F_i^+ [7] itself is also given by an elaboration into F_{co}, a System F-like language with products. Unlike F_i^+, F_{co} has no subtyping or intersection types, and it has a conventional operational semantics. The main reason for F_i^+ to use elaboration is that F_i^+ has a *type-dependent* semantics: types may affect the runtime behavior of a program. The elaboration semantics for F_i^+ seems like a natural choice, since this is commonly seen in various other type-dependent languages and calculi. For instance, the semantics of type-dependent languages with *type classes* [43], *Scala-style implicits* [29] or *gradual typing* [40] all usually adopt an elaboration approach. In contrast, in the past, more conventional direct formulations using an operational semantics have been avoided for languages with a type-dependent semantics. The appeals of the elaboration semantics are simple type-safety proofs, and the fact that they directly offer an implementation technique over conventional languages without a type-dependent semantics.

There are also important drawbacks when using an elaboration semantics. One of them is simply that more infrastructure is needed for a target language (such as F_{co}) and its associated semantics and metatheory. Moreover, the elaboration semantics is indirect, and to understand the semantics of a program, we must first translate it to the target language (which may be significantly different from the source) and then reason in terms of the target. More importantly, besides type-safety, another property that is often desirable for an elaboration semantics is *coherence* [35]. Many elaboration semantics are non-deterministic: the same source program can elaborate into different target programs. If those different programs have a different semantics, then this is problematic, as it would imply that the source language would have a non-deterministic or ambiguous semantics. Coherence ensures that even if the same program elaborates to different target expressions, the different target expressions are semantically equivalent, eventually evaluating to the same result.

For some languages, including F_i^+, proving coherence is highly non-trivial and hard to scale to common programming language features. For the original F_i^+, the proof of coherence comes at the cost of simple features such as *recursion* and *impredicative polymorphism*. The

proof of coherence for F_i^+ is based on a logical relation called *canonicity* [6]. Together with a notion of contextual equivalence, the two techniques are used to prove coherence. The use of logical relations is a source of complexity in the proof and the reason why recursion and impredicative polymorphism have not been supported. For recursion, in principle, the use of a more sophisticated *step-indexed* logical relation [3] may enable a proof of coherence, at the cost of some additional complexity. However, due to the extra complexity, this was left for future work. For impredicative polymorphism, Bi et al. [7] identified important technical challenges, and it is not known if the proof can be extended with such a feature.

The absence of recursion and impredicative polymorphism creates a gap between theory and practice, since CP fundamentally relies on them. Moreover, the proofs of correctness of CP rely on the assumption that F_i^+ with recursion and impredicative polymorphism would preserve all the properties of F_i^+. Impredicative polymorphism is needed in CP to allow the types of traits with polymorphic methods to be used as type parameters for other polymorphic functions. Recursion is needed in CP because the denotational semantics uses fixpoint operators to instantiate traits. In addition, the fixpoint operators must be *lazy*; otherwise, self-references can easily trigger non-termination. Therefore, a *call-by-name* (CBN) semantics is more natural and also assumed in the CP encoding of traits. However, the semantics of the F_{co} calculus is *call-by-value* (CBV) and, by inheritance, the elaboration semantics of F_i^+ has a CBV semantics as well.

This paper presents a new formulation of F_i^+ based on a *type-directed operational semantics* (TDOS) [20]. The TDOS approach has recently been proposed to model the semantics of languages with disjoint intersection types (but without polymorphism). Although F_i^+ is not a new calculus, we revise its formulation significantly in this paper. Our new formulation of F_i^+ is different from the original one in three aspects. Firstly, the semantics of the original F_i^+ is given by elaborating to F_{co}, while our semantics for F_i^+ is a direct operational semantics. Secondly, our new formulation of F_i^+ supports recursion and impredicative polymorphism. Finally, we employ a call-by-name evaluation strategy.

Our work shows that the TDOS approach can be extended to languages with disjoint polymorphism and model the complete F_i^+ calculus with recursion and impredicative polymorphism. Moreover, there is no need for a coherence proof. Instead, we can simply prove that the semantics is *deterministic*. The proof of determinism uses only simple reasoning techniques, such as straightforward induction, and is able to handle problematic features such as recursion and impredicative polymorphism. Thus, this removes the gap between theory and practice and validates the original proofs of correctness for the CP language. Figure 1 contrasts the differences in terms of proofs and implementation of CP using Zhang et al.'s original work and our own work. We formalized the TDOS variant of the F_i^+ calculus, together with its type-soundness and determinism proof in the Coq proof assistant. Moreover, we have a new implementation of CP based on our new reformulation of F_i^+.

In summary, the contributions of this work are:

- **CBN F_i^+ with recursion and impredicative polymorphism.** This paper presents a CBN variant of F_i^+ extended with recursion and impredicative polymorphism.
- **Determinism and type-soundness for F_i^+ using a TDOS.** We prove the type-soundness and determinism of F_i^+ using a direct TDOS. These proofs validate the proofs of correctness previously presented for CP by Zhang et al. [46].
- **Technical innovations.** Our formulation of F_i^+ has various technical innovations over the original one, including a new formulation of subtyping using splittable types [22] and more flexible term applications.

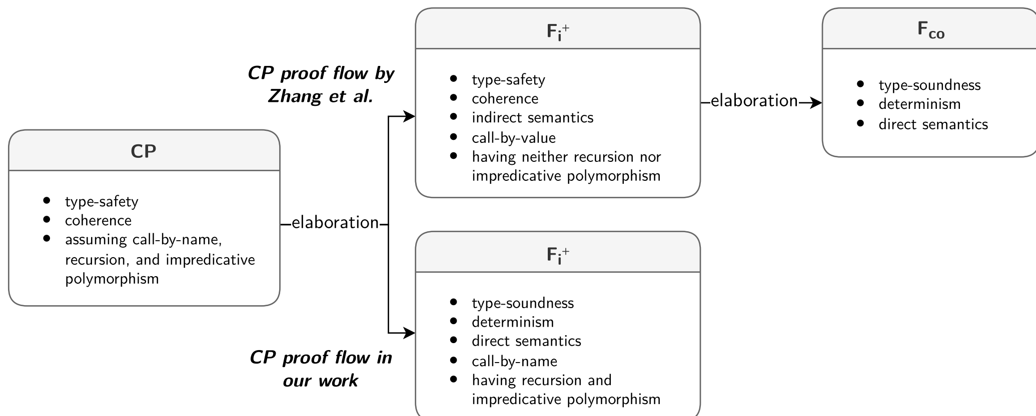


Figure 1 Contrasting the flow of results for CP using the original formulation, and our work.

- **Implementation and Mechanical formalization.** We formalized the TDOS variant of the F_i^+ calculus, together with its type-soundness and determinism proof in the Coq proof assistant. We also have a new implementation of CP built on top of a TDOS formulation of F_i^+ available at `https://plground.org`. The full Coq formalization and the extended version of this paper are available at:

`https://github.com/andongfan/CP-Foundations`

2 Motivations and Technical Innovations

In this section, we introduce Compositional Programming by example and show how CP traits elaborate to F_i^+ expressions. After that, we will discuss the practical issues that motivate us to reformulate F_i^+, as well as technical challenges and innovations.

2.1 Compositional Programming by Example

To demonstrate the capabilities of Compositional Programming, we show how to solve a variant of the *Expression Problem* [42] in the CP language. Our solution is adapted from the original one by Zhang et al. [46]. In this variant, in addition to the usual challenge of extensibility in multiple directions, we also consider the problem of *context evolution* [25, 37], so the interpreter may require different contextual information for different features of the interpreter. The CP language allows a modular solution to both challenges, which also illustrates some key features in Compositional Programming, including *first-class traits* [5], *nested composition* [6], and *disjoint polymorphism* [2].

Examples are based on a simple expression language, and the goal is to perform various operations over it, such as evaluation and free variable bookkeeping. The expression language consists of numbers, addition, variables, and let-bindings. Besides CP code, we also provide analogous Haskell code in the initial examples so that readers can connect them with existing concepts in functional languages.

Compositional interfaces. First, we define the compositional interface for numeric literals and addition. The compositional interface at the top of Figure 2a is similar to Haskell's algebraic data type at the top of Figure 2b. `Exp` is a special kind of type parameter in CP called a *sort*, which serves as the return type of both constructors `Lit` and `Add`. Sorts will

```
type NumSig<Exp> = {
  Lit : Int → Exp;
  Add : Exp → Exp → Exp;
};

type Eval Ctx = { eval : Ctx → Int };
evalNum Ctx = trait implements
    NumSig<Eval Ctx> ⇒ {
  (Lit     n).eval _    = n;
  (Add e1 e2).eval ctx =
     e1.eval ctx + e2.eval ctx;
};
```

(a) CP code.

```
data Exp where
  Lit :: Int → Exp
  Add :: Exp → Exp → Exp


type Eval ctx = ctx → Int

eval :: Exp → Eval ctx
eval (Lit     n) _    = n
eval (Add e1 e2) ctx =
   eval e1 ctx + eval e2 ctx
```

(b) Haskell counterpart.

Figure 2 Initial expression language: numbers and addition.

be instantiated with concrete representations later. Internally, sorts are handled differently from normal type parameters [46]. In accordance with the compositional interface, we can then define how to evaluate the expression language.

Polymorphic contexts. As shown in the middle of Figure 2a, the type `Eval` declares a method `eval` that takes a context and returns an integer. `Ctx` is a type parameter that can be instantiated later, enabling particular traits to assume particular contextual information for the needs of various features. The technique is called *polymorphic contexts* [46] in Compositional Programming.

Compositional traits. The trait `evalNum` in Figure 2a is parametrized by a type parameter `Ctx`. Note that, in CP, type parameters always start with a capital letter, while regular parameters are lowercase. The trait `evalNum` implements the compositional interface `NumSig` by instantiating it with the sort `Eval Ctx`. *Traits* are the basic reusable unit in CP, which are usually type-checked against compositional interfaces. In this trait, we use a lightweight syntax called *method patterns* to define how to evaluate different expressions. Such a definition is analogous to pattern matching in Figure 2b. Since `Lit` and `Add` do not need to be conscious of any information in the context, the type parameter `Ctx` is unconstrained. The only thing that we can do to the polymorphic context is either to ignore it (like in `Lit`) or to pass it to recursive calls (like in `Add`).

More expressions. Adding more constructs to the expression language is awkward in Haskell because algebraic data types are *closed*. However, language components can be modularly declared in CP. Two new constructors, `Let` and `Var`, are declared in the second compositional interface `VarSig`, as shown in Figure 3. Then the two traits implement `VarSig` using method patterns for the new constructors. Since the two new expressions need to inspect or update some information in the context, we expose the appropriate `Env` part to `evalVar`, while the remaining context is kept polymorphic. This is achieved with the *disjointness constraint* [2] `Ctx*Env` in `evalVar`. A disjointness constraint denotes that the type parameter `Ctx` is disjoint to the type `Env`. In other words, types that instantiate `Ctx` cannot overlap with the type `Env`. Also note that the notation `{ ctx with env = ... }` denotes a *polymorphic record update* [9]. In the code for let-expressions, we need to update the environment in the recursive calls to extend it with a new entry for the let-variable.

```
type VarSig<Exp> = {
  Let : String → Exp → Exp → Exp;
  Var : String → Exp;
};

type Env = { env : String → Int };
evalVar (Ctx*Env) = trait implements VarSig<Eval (Env&Ctx)> ⇒ {
  (Let s e1 e2).eval ctx = e2.eval
    { ctx with env = insert s (e1.eval ctx) ctx.env };
  (Var       s).eval ctx = lookup s ctx.env;
};
```

Figure 3 Adding more expressions: variables and let-bindings.

```
type FV = { fv : [String] };
fv = trait implements ExpSig<FV> ⇒ {
  (Lit       n).fv = [];
  (Add    e1 e2).fv = union e1.fv e2.fv;
  (Let s e1 e2).fv = union e1.fv (delete s e2.fv);
  (Var       s).fv = [s];
};

evalWithFV (Ctx*Env) = trait implements ExpSig<FV ⇒ Eval (Env&Ctx)> ⇒ {
  (Lit       n).eval _   = n;
  (Add    e1 e2).eval ctx = e1.eval ctx + e2.eval ctx;
  (Let s e1 e2).eval ctx = if elem s e2.fv
    then e2.eval { ctx with env = insert s (e1.eval ctx) ctx.env }
    else e2.eval ctx;
  (Var       s).eval ctx = lookup s ctx.env;
};
```

Figure 4 Adding more operation: free variable bookkeeping and another version of evaluation.

Intersection types. Independently defined interfaces can be composed using *intersection types*. For example, `ExpSig` below is an intersection of `NumSig` and `VarSig`, containing all of the four constructors:

```
type ExpSig<Exp> = NumSig<Exp> & VarSig<Exp>;
--               = { Lit : ...; Add : ...; Let : ...; Var : ... };
```

More operations. Not only can expressions be modularly extended, but we can easily add more operations. In Figure 4, a new trait `fv` modularly implements a new operation that records free variables in an expression. Here, `union` and `delete` are two library functions for arrays. The modular definition of `fv` is quite natural in functional programming, but it is hard in traditional object-oriented programming. We have to modify the existing class definitions and supplement them with a method. This is typical of the well-known Expression Problem. In summary, we have shown that Compositional Programming can solve both dimensions of this problem: adding expressions and operations.

Dependency injection. Besides the Expression Problem, Figure 4 also shows another significant feature of CP: dependency injection. In `evalWithFV`, a new implementation of evaluation is defined with a dependency on free variables. The method pattern for `Let` will check if `s` appears as a free variable in `e2`. If so, it evaluates `e1` first as usual; otherwise, we do not need to do any computation or update the environment since `s` is not used at all. Note that the compositional interface `ExpSig` is instantiated with two types separated by a fat arrow (⇒) (⇒ was originally denoted by `%` in Zhang et al.'s implementation of CP). `FV` on the left-hand side is the dependency of `evalWithFV`. In other words, the definition of `evalWithFV` depends on another trait that implements `ExpSig<FV>`. The static type checker of CP will check this fact later at the point of trait instantiation. With such dependency injection, we can call `e2.fv` even if `evalWithFV` does not have an implementation of `fv`. In other words, `evalWithFV` depends only on the interface of `fv` (the type `FV`), but not any concrete implementation.

Self-type annotations. Before we show how to perform the new version of the evaluation over the whole expression language, we want to create a repository of expressions for later use. We expect that these expressions are unaware of any concrete operation, so we use a polymorphic `Exp` type to denote some abstract type of expressions. The code that creates the repository of expressions is[1]:

```
repo Exp = trait [self : ExpSig<Exp>] ⇒ {
  num = Add (Lit 4) (Lit 8);
  var = Let "x" (Lit 4) (Let "y" (Lit 8) (Add (Var "x") (Var "y")));
};
```

To make constructors available from the compositional interface, we add a *self-type annotation* to the trait `repo`. The self type annotation `[self : ExpSig<Exp>]` imposes a requirement that the `repo` should finally be merged with some trait implementing `ExpSig<Exp>`. This requirement is also statically enforced by the static type checker of CP. This is the second mechanism in Compositional Programming to modularly inject dependencies.

Nested trait composition. With the language components ready, we can compose them using the merge operator [14], which in the CP language is denoted as a single comma (`,`). First, we show how to compose the old version of the evaluation:

```
exp = new repo @(Eval Env) , evalNum @Env , evalVar @Top;
exp.var.eval { env = empty }  --> 12
```

Since the context has evolved after we add variables, we pass different type arguments to the two traits to make the final context consistent. The final context type is `Env`, so we pass `Env` to `evalNum` and `Top` to `evalVar`. Type arguments are prefixed by `@` in CP. A more interesting example is to merge the new version of evaluation with free variable bookkeeping:

```
exp' = new repo @(Eval Env & FV) , evalWithFV @Top , fv;
exp'.var.eval { env = empty }  --> 12
```

After the trait composition, both operations (`eval` and `fv`) are available for expressions that are built with the four constructors (`Lit`, `Add`, `Let`, and `Var`). Note that here `fv` satisfies the dependency of `evalWithFV`. If no implementation of the type `FV` is present in the

[1] In Zhang et al.'s original work [46], the **`new`** operator must be added before every constructor. However, our new implementation will implicitly insert **`new`** (see Section 2.2 for details).

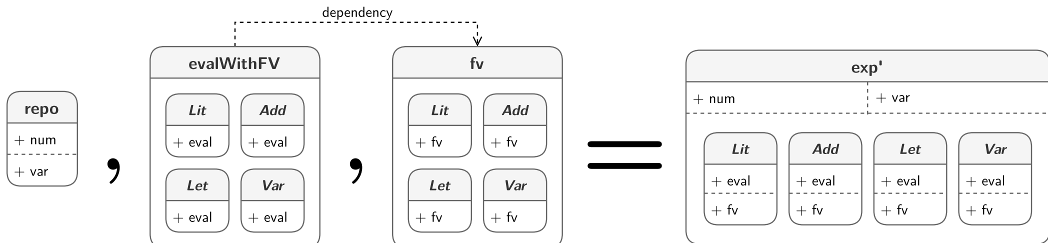


Figure 5 Visualization of nested composition.

composition, there would be a type error, since the requirement for `evalWithFV` would not be satisfied. The composition of the three traits is *nested* because the two methods nested in the four constructors are composed, as visualized in Figure 5. With nested trait composition, the Expression Problem is elegantly solved in Compositional Programming. Moreover, we allow context evolution using a relatively simple way with polymorphic contexts.

Impredicative polymorphism. Another feature of CP is that it allows the creation of objects with polymorphic methods, similar to most OOP languages with generics where classes can contain polymorphic methods (like Java). However, for this to work properly, CP must support *impredicative polymorphism* (the ability to instantiate type parameters with polymorphic types) as System F does. For example, consider:

```
type Poly = { id : forall A. A → A };
idTrait = trait implements Poly ⇒ { id = ΛA. \(x:A) → x };

(new idTrait).id @Poly  -- impredicative
```

While accepted by our variant of CP and F_i^+, such polymorphic instantiations are forbidden in the original formulation of F_i^+.

2.2 Elaborating CP to F_i^+

Under the surface of CP, the foundation for Compositional Programming is the F_i^+ calculus. We present the key features in F_i^+ and take a closer look at the connection between CP and F_i^+ expressions. Here we focus on the elaboration of traits, which are the most important aspect of this paper. We refer curious readers to the work by Zhang et al. [46] for the full formulation of the type-directed elaboration of CP.

Key features of the F_i^+ calculus. F_i^+ is basically a variant of System F [17, 33] extended with intersection types and a merge operator. In the F_i^+ calculus, we denote the merge operator with a double comma (,,) (instead of the single comma notation in CP), following the original notation proposed by Dunfield [14]. The merge operator allows us to introduce terms of intersection types. For example, $1\,,,\,\mathsf{true}$ is a term of type $\mathsf{Int}\,\&\,\mathsf{Bool}$. Moreover, record concatenation, which is used to encode multi-field traits, is encoded as merges of records in F_i^+. Thus, multi-field records are represented as merges of multiple single-field records. However, to ensure determinism of operational semantics, not all terms can be merged with each other. We impose a disjointness check when typing merges: a merge can only type check when the types of the terms being merged are disjoint, ensuring that every part in a merge can be distinguished by its type. For traits, for example, the disjointness restriction ensures that traits cannot have two fields/methods with the same name `m` and overlapping types, which could otherwise lead to ambiguity when doing method lookup. Here,

we show an example of ambiguity if there is no disjointness check. With intersection types, both $A \,\&\, B \leq A$ and $A \,\&\, B \leq B$ are valid. Therefore, a merge $1\,,,\,2$ of type $\mathsf{Int} \,\&\, \mathsf{Int}$ can be typed with Int, but at runtime, two different values of type Int are found. Thus, an expression such as $(1\,,,\,2)+1$ could evaluate to either 2 or 3. Since we wish for a deterministic semantics, we use disjointness to prevent such forms of ambiguity. On the other hand, $(1\,,,\,\mathsf{true})+1$ type checks because Int and Bool are disjoint, and it evaluates to 2 unambiguously. A disjointness constraint can also be added to a type variable in a System F-style polymorphic type, such as $\forall X * \mathsf{Int}.\, X \,\&\, \mathsf{Int}$. Moreover, to support unrestricted intersection types like $\mathsf{Int} \,\&\, \mathsf{Int}$, the disjointness check is relaxed to *consistency* for certain terms, so that merges with duplications like $1\,,,\,1$ are allowed.

Elaborating traits into F_i^+. The elaboration of traits is inspired by Cook's denotational semantics of inheritance [12]. To use a concrete example, we revisit the trait `repo` defined in Section 2.1. Both the creation and instantiation of traits are included in the definition of `repo`:

```
repo Exp = trait [self : ExpSig<Exp>] ⇒ {
  num = Add (Lit 4) (Lit 8);
  var = ...
};
```

The CP code above is elaborated to corresponding F_i^+ code of the form:

```
repo = ΛExp. λ(self : ExpSig<Exp>).
  let $Lit = self.Lit in let $Add = self.Add in
  let $Let = self.Let in let $Var = self.Var in
    { num = fix self:Exp. $Add (fix self:Exp. $Lit 4 self)
                               (fix self:Exp. $Lit 8 self) self } ,,
    { var = ... };
```

The type parameter `Exp` in the `repo` trait is expressed by a System-F-style type lambda $(\Lambda X.\, e)$. Note that CP employs a form of syntactic sugar for constructors to allow concise use of constructors and avoid explicit uses of **`new`**. The source code `Add (Lit 4) (Lit 8)` is first expanded into `new $Add (new $Lit 4) (new $Lit 8)`, which insert **`new`** operators. Next we describe the elaboration process of creating and instantiating traits:

- **Creation of traits**: A **`trait`** is elaborated to a *generator* function whose parameter is a self-reference (like `self` above) and whose body is a record of methods;
- **Instantiation of traits**: The **`new`** construct is used to instantiate a trait. Uses of **`new`** are elaborated to a *fixpoint* which applies the elaborated trait function to a self-reference. In the definition of the field `num` there are three elaborations of **`new`**. For instance, the CP code `new $Lit 4` corresponds to the F_i^+ code `fix self:Exp. $Lit 4 self`.

It is clear now that our trait encoding is heavily dependent on recursion, due to the self-references employed by the encoding. However, the original F_i^+ [7] does not support recursion, which reveals a gap between theory and practice.

2.3 The Gap Between Theory and Practice

Our primary motivation to reformulate F_i^+ is to bridge the gap between theory and practice. The original formulation of F_i^+ lacks recursion, impredicative polymorphism and uses the traditional call-by-value (CBV) evaluation strategy. However, the recent work of CP assumes a different variant of F_i^+ that is equipped with fixpoints and the call-by-name (CBN) evaluation. It is worthwhile to probe into the causes of such differences.

Non-triviality of coherence. Recursion is essential for general-purpose computation in programming. More importantly, our encoding of traits requires recursion. For example, `new e` is elaborated to `fix self. e self`. However, adding recursion to the original version of F_i^+ turns out to be highly non-trivial. The original F_i^+ is defined using an elaboration semantics. A fundamental property of F_i^+ is *coherence* [35], which states that the semantics is unambiguous. Coherence is non-trivial due to the presence of the merge operator [14]. To prove coherence, a logical relation, called *canonicity* [7], is used to reason about contextual equivalence in the original work of F_i^+. For example, with contextual equivalence, we can show that the two possible elaborations for the same F_i^+ source expression into F_{co} are contextually equivalent:

$$1 : \mathsf{Int} \,\&\, \mathsf{Int} : \mathsf{Int} \;\rightsquigarrow\; \mathsf{fst}\,(1,1) \qquad\qquad 1 : \mathsf{Int} \,\&\, \mathsf{Int} : \mathsf{Int} \;\rightsquigarrow\; \mathsf{snd}\,(1,1)$$

Two typing derivations lead to two elaborations in this example, which pick different sides of the merge. However, both elaborated expressions will be reduced to 1 eventually.

Unfortunately, the proof technique for coherence based on logical relations does not immediately scale to recursive programs and programs with impredicative polymorphism. A possible solution, known from the research of logical relations, is to move to a more sophisticated *step-indexed* form of logical relations [1]. However, this requires a major reformulation of the proofs and metatheory of the original F_i^+, and it is not clear whether additional challenges would be present in such an extension. Thus, the lack of the two features in the theory of the original F_i^+ remains a serious limitation since only terminating programs and predicative polymorphism are considered. In other words, we cannot encode traits as presented in Section 2.2 in the original F_i^+. To get around this issue and enable the encoding of traits, Zhang et al. [46] simply assumed an extension of F_i^+ with recursion and their proof of coherence for CP was done under the assumption that the original F_i^+ with recursion was coherent or deterministic.

Our work rectifies this gap in the theory of Compositional Programming and the CP language. We reformulate F_i^+ using a direct type-directed operational semantics [22] that allows recursion and prove that the semantics is deterministic. Thus, our reformulation of F_i^+ can serve as a target language to encode traits and validate the proofs of the elaboration of CP in terms of F_i^+ with recursion. In addition, our approach gives a semantics to F_i^+ directly, instead of relying on an indirect elaboration semantics to a System F-like language.

Evaluation strategies. Most mainstream programming languages use CBV, but CBN is a more natural evaluation strategy for object encodings such as Cook's denotational semantics of inheritance. As stated by Bruce et al. in their work on object encodings [8]:

> *"Although we shall perform conversion steps in whatever order is convenient for the sake of examples, we could just as well impose a call-by-name reduction strategy. (Most of the examples would diverge under a call-by-value strategy. This can be repaired at the cost of some extra lambda abstractions and applications to delay evaluation at appropriate points.)"*

In our elaboration of traits, we adopt a similar approach to object encodings. For example, consider the following CP expression:

```
type A = { l1 : Int; l2 : Int };
new (trait [self : A] ⇒ { l1 = 1; l2 = self.l1 })
```

which is elaborated to the following (slightly simplified) F_i^+ expression:

$$\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}$$

The **trait** expression is elaborated to a function, and the **new** expression turns the function into a fixpoint. Unfortunately, this expression terminates under CBN but diverges under CBV. If evaluated under CBV, the variable *self* will be evaluated repeatedly, despite the fact that only $\mathit{self}.l_1$ is used:

$$\begin{aligned}
& \textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\} \\
\hookrightarrow\ & \{l_1 = 1\}\,,,\,\{l_2 = (\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}).l_1\} \\
\hookrightarrow\ & \{l_1 = 1\}\,,,\,\{l_2 = (\{l_1 = 1\}\,,,\,\{l_2 = (\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}).l_1\}).l_1\} \\
\hookrightarrow\ & \cdots
\end{aligned}$$

We may tackle the problem of non-termination by wrapping self-references in *thunks*, but CBN provides a simpler and more natural way. In our CBN formulation of F_i^+, $\{l = e\}$ is already a value (instead of $\{l = v\}$), so we do not need to further evaluate e:

$$\begin{aligned}
& \textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\} \\
\hookrightarrow\ & \{l_1 = 1\}\,,,\,\{l_2 = (\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}).l_1\}
\end{aligned}$$

The l_2 field is further evaluated only when a record projection is performed:

$$\begin{aligned}
& (\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}).l_2 \\
\hookrightarrow\ & (\{l_1 = 1\}\,,,\,\{l_2 = (\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}).l_1\}).l_2 \\
\hookrightarrow\ & (\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}).l_1 \\
\hookrightarrow\ & (\{l_1 = 1\}\,,,\,\{l_2 = (\textbf{fix}\ \mathit{self}:A.\,\{l_1 = 1\}\,,,\,\{l_2 = \mathit{self}.l_1\}).l_1\}).l_1 \\
\hookrightarrow\ & 1
\end{aligned}$$

This example illustrates how our new CBN formulation of F_i^+ avoids non-termination of trait instantiation.

2.4 Technical Challenges and Innovations

The main novelty of our reformulation of F_i^+ is the use of a type-directed operational semantics [20] instead of an elaboration semantics. With a TDOS, adding recursion and impredicative polymorphism to our proof of determinism is trivial. Our work is an extension of the λ_i^+ calculus [22] which adapts the TDOS approach. We also follow the subtyping algorithm design in λ_i^+. While λ_i^+ supports BCD-style distributive subtyping [4], the addition of disjoint polymorphism does bring some technical challenges. Moreover, there are some smaller changes to F_i^+ that enable us to type-check more programs and improve the design of the original F_i^+. We will give an overview of the technical challenges and innovations next.

The role of casting. A merge like $1\,,,\,\mathsf{true}$ has multiple meanings under different types (e.g. Int or Bool). Eventually, we have to extract some components via the elimination of merges, which is a key issue when designing a direct operational semantics for a calculus with the merge operator. A non-deterministic semantics could allow $e_1\,,,\,e_2 \hookrightarrow e_1$ and $e_1\,,,\,e_2 \hookrightarrow e_2$ without any constraints, at the cost of losing both type preservation and determinism [14].

To obtain a non-ambiguous and type-safe semantics, we follow the TDOS approach [20]: which uses (up)casts to ensure that values have the right form during reduction. In a TDOS, there is a casting relation, which is used in the reduction rule for annotated values:

$$\frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \textsc{Step-annov}$$

Casting enables us to drop certain parts from a term (e.g., $1\,,,\,\mathsf{true} \hookrightarrow_{\mathsf{Int}} 1$). Very often, it is necessary for us to do so to satisfy the disjointness constraint. Consider a function $\lambda x{:}\,\mathsf{Int}.\,x\,,,\,\mathsf{false}$. For its body to be well-typed, x cannot contain a boolean. Hence, when the function is applied to $1\,,,\,\mathsf{true}$, we cannot directly substitute the argument in. Instead, it is wrapped by (and later cast to) Int to resolve the potential conflict.

$$\begin{aligned} & ((\lambda x{:}\,\mathsf{Int}.\,x\,,,\,\mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \to \mathsf{Int}\,\&\,\mathsf{Bool})\,(1\,,,\,\mathsf{true}) \\ \hookrightarrow\;& ((1\,,,\,\mathsf{true}) : \mathsf{Int}\,,,\,\mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \\ \hookrightarrow\;& (1\,,,\,\mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \\ \hookrightarrow\;& 1\,,,\,\mathsf{false} \end{aligned}$$

TDOS and function annotations. In casting, values in a merge are selected based on type information. In the absence of runtime type-checking, we need to know the type of input value syntactically to match it with the target type. Thus, functions must be accompanied by type annotations. The previous work λ_i^+ [22] defines the syntax of functions like $\lambda x.\,e : A \to C$. While the original argument type A is always kept during reduction, λ_i^+'s casting relation may generate a value that has a proper subtype of the requested type: $\lambda x.\,e : A \to C \hookrightarrow_{B_1 \to B_2} \lambda x.\,e : A \to B_2$. We make casting more precise with a more liberal syntax in F_i^+. We allow bare abstractions $\lambda x{:}\,A.\,e$ while λ_i^+ does not. Our casting relation requires lambdas to be annotated $(\lambda x{:}\,A.\,e) : B$, but the full annotation B does not have to be a function type. For example, $(\lambda x{:}\,\mathsf{Int}.\,x\,,,\,\mathsf{true}) : (\mathsf{Int} \to \mathsf{Int})\,\&\,(\mathsf{Int} \to \mathsf{Bool})$ still acts as a function, and is equivalent to $(\lambda x{:}\,\mathsf{Int}.\,x\,,,\,\mathsf{true}) : \mathsf{Int} \to \mathsf{Int}\,\&\,\mathsf{Bool}$.

Algorithmic subtyping with disjoint polymorphism. F_i^+ extends BCD-style distributive subtyping [4] to disjoint polymorphism. $\forall X * \mathsf{Int}.\,X\,\&\,\mathsf{Int}$ represents the intersection of some type X and Int assuming X is disjoint to Int. Like arrows or records, such universal types distribute over intersections. Hence, $(\forall X * \mathsf{Int}.\,X)\,\&\,(\forall X * \mathsf{Int}.\,\mathsf{Int})$ is a subtype of $\forall X * \mathsf{Int}.\,X\,\&\,\mathsf{Int}$. A well-known challenge in supporting distributivity in the BCD-style subtyping is to obtain an algorithmic formulation of subtyping. There have been many efforts to eliminate the explicit transitivity rule to obtain an algorithmic formulation [26, 31, 39]. Compared with the original F_i^+ [7], we employ a different subtyping algorithm design, using *splittable types* [21]. This approach employs a type-splitting operation ($B \lhd A \rhd C$) that converts a given type A to an equivalent intersection type $B\,\&\,C$, for example, $A \to B_1\,\&\,B_2$ is split to $A \to B_1$ and $A \to B_2$. The subtyping algorithm uses type splitting whenever an intersection type is expected in the conventional algorithm for subtyping without distributivity, and therefore handles distributivity smoothly and modularly. For space reasons, the novel algorithmic subtyping approach is discussed in the extended version of the paper.

Enhanced subtyping and disjointness with more top-like types. Unlike previous systems with disjoint polymorphism [2, 7], we add a context in subtyping judgments to track the disjointness assumption $X * A$ whenever we open a universal type $\forall X * A.\,B$, similar to the subtyping with F-bounded quantification. The extra information enhances our subtyping: we know a type must be a supertype of Top, if it is disjoint with Bot. This also fixes the following broken property in the original F_i^+, as we now have more types that are *top-like*.

▶ **Definition 2.1** (Disjointness specification). *If A is disjoint with B, any common supertypes they have must be equivalent to* Top.

Keeping this property is necessary for us to obtain a deterministic operational semantics. Meanwhile, we prove our subtyping and disjointness relations are decidable in Coq. Note that in the original F_i^+, the decidability of the two relations was proved manually, although the rest of the proof was mechanized.

3 The F_i^+ Calculus and Its Operational Semantics

This section introduces the F_i^+ calculus, including its static and dynamic semantics.

3.1 Syntax

The syntax of F_i^+ is as follows:

Types	$A, B, C ::= X \mid \mathsf{Int} \mid \mathsf{Top} \mid \mathsf{Bot} \mid A \,\&\, B \mid A \to B \mid \forall X * A.\, B \mid \{l : A\}$
Checkable terms	$p ::= \lambda x : A.\, e \mid \Lambda X.\, e \mid \{l = e\}$
Expressions	$e ::= p \mid x \mid i \mid \top \mid e : A \mid e_1 \,,,\, e_2 \mid \textbf{fix}\ x : A.\, e \mid e_1\, e_2 \mid e\, A \mid e.l$
Values	$v ::= p \mid p : A \mid i \mid \top \mid v_1 \,,,\, v_2$
Term contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Type contexts	$\Delta ::= \cdot \mid \Delta, X * A$

Types. Types include the Top type and the uninhabited type Bot. Intersection types are created with $A \,\&\, B$. Disjoint polymorphism, a key feature of F_i^+, is based on universal types with a disjointness quantifier $\forall X * A.\, B$, expressing that the type variable X is bound inside B and disjoint to type A. $\{l : A\}$ denotes single-field record types, where l is the record label. Multi-field record types are desugared to intersections of single-field ones [36]:

$$\{l_1 : A_1;\ \ldots;\ l_n : A_n\} \triangleq \{l_1 : A_1\} \,\&\, \cdots \,\&\, \{l_n : A_n\}$$

Expressions. As we will explain later with the typing rules, some expressions do not have an inferred type (or principal type), including lambda abstractions, type abstractions, and single-field records. We use metavariable p to represent these expressions, which with optional annotations, are values. Also, note that expressions inside record values do not have to be a value since our calculus employs call-by-name. The merge operator $,,$ composes two expressions to make a term of an intersection type. The top value $\top$ can be viewed as a merge of zero elements. Fixpoint expressions $\textbf{fix}\ x : A.\, e$ construct recursive programs. The type annotation A denotes the type of x as well as the whole expression. Like record types, multi-field records are desugared to merges of single-field ones:

$$\{l_1 = e_1;\ \ldots;\ l_n = e_n\} \triangleq \{l_1 = e_1\} \,,,\, \ldots \,,,\, \{l_n = e_n\}$$

Contexts. We have two contexts: Γ tracks the types of term variables; Δ tracks the disjointness information of type variables, which follows the original design of F_i^+. We use $\Delta \vdash A$, $\vdash \Delta$, and $\Delta \vdash \Gamma$ judgments for the type well-formedness and the context well-formedness (defined in the extended version of the paper). For multiple type well-formedness judgments, we combine them into one, i.e., $\Delta \vdash A, B$ means $\Delta \vdash A$ and $\Delta \vdash B$.

$\boxed{\Delta \vdash A <: B}$ *(Declarative Subtyping)*

$$\textsc{DS-refl}\quad \frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: A}$$

$$\textsc{DS-trans}\quad \frac{\Delta \vdash A <: B \qquad \Delta \vdash B <: C}{\Delta \vdash A <: C}$$

$$\textsc{DS-top}\quad \frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: \mathsf{Top}}$$

$$\textsc{DS-bot}\quad \frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash \mathsf{Bot} <: A}$$

$$\textsc{DS-and}\quad \frac{\Delta \vdash A <: B \qquad \Delta \vdash A <: C}{\Delta \vdash A <: B \,\&\, C}$$

$$\textsc{DS-andL}\quad \frac{\vdash \Delta \qquad \Delta \vdash A, B}{\Delta \vdash A \,\&\, B <: A}$$

$$\textsc{DS-andR}\quad \frac{\vdash \Delta \qquad \Delta \vdash A, B}{\Delta \vdash A \,\&\, B <: B}$$

$$\textsc{DS-arrow}\quad \frac{\Delta \vdash A_2 <: A_1 \qquad \Delta \vdash B_1 <: B_2}{\Delta \vdash A_1 \to B_1 <: A_2 \to B_2}$$

$$\textsc{DS-distArrow}\quad \frac{\vdash \Delta \qquad \Delta \vdash A, B, C}{\Delta \vdash (A \to B) \,\&\, (A \to C) <: A \to B \,\&\, C}$$

$$\textsc{DS-topArrow}\quad \frac{\vdash \Delta}{\Delta \vdash \mathsf{Top} <: \mathsf{Top} \to \mathsf{Top}}$$

$$\textsc{DS-rcd}\quad \frac{\Delta \vdash A <: B}{\Delta \vdash \{l : A\} <: \{l : B\}}$$

$$\textsc{DS-distRcd}\quad \frac{\vdash \Delta \qquad \Delta \vdash A, B}{\Delta \vdash \{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\}}$$

$$\textsc{DS-topRcd}\quad \frac{\vdash \Delta}{\Delta \vdash \mathsf{Top} <: \{l : \mathsf{Top}\}}$$

$$\textsc{DS-all}\quad \frac{\Delta \vdash A_2 <: A_1 \qquad \Delta, X * A_2 \vdash B_1 <: B_2}{\Delta \vdash \forall X * A_1.\, B_1 <: \forall X * A_2.\, B_2}$$

$$\textsc{DS-topAll}\quad \frac{\vdash \Delta}{\Delta \vdash \mathsf{Top} <: \forall X * \mathsf{Top}.\, \mathsf{Top}}$$

$$\textsc{DS-distAll}\quad \frac{\vdash \Delta \qquad \Delta \vdash A \qquad \Delta, X * A \vdash B_1, B_2}{\Delta \vdash (\forall X * A.\, B_1) \,\&\, (\forall X * A.\, B_2) <: \forall X * A.\, (B_1 \,\&\, B_2)}$$

$$\textsc{DS-topVar}\quad \frac{X * A \in \Delta \qquad \Delta \vdash A <: \mathsf{Bot}}{\Delta \vdash \mathsf{Top} <: X}$$

Figure 6 Declarative subtyping rules.

3.2 Subtyping

Figure 6 shows our subtyping relation, which extends BCD-style subtyping [4] with disjoint polymorphism, records, and the bottom type. Compared with the original F_i^+, we add a context to track type variables and their disjointness information. The context not only ensures the well-formedness of types, but is also important to our new rule DS-topVar. An equivalence relation (Definition 3.1) is defined on types that are subtype of each other. These equivalent types can be converted back and forth without loss of information.

▶ **Definition 3.1** (Type equivalence). $\Delta \vdash A \sim B \triangleq \Delta \vdash A <: B$ *and* $\Delta \vdash B <: A$.

For functions (rule DS-arrow) and disjoint quantifications (rule DS-all), subtyping is covariant in positive positions and contravariant in negative positions. The intuition is that type abstractions of the more specific type (subtype) should have a *looser* disjointness constraint for the parameter type. $\forall X * \mathsf{Top}.\, A$ denotes that there is no constraint on X, since Top is disjoint to all types. On the contrary, Bot is the strictest constraint. It is useful in types like $\forall X * \{l : \mathsf{Bot}\}.\, A$ which expresses that X does not contain any informative field of label l [44]. For intersection types, rules DS-andL, DS-andR, and DS-and axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. As a typical characteristic of BCD-style subtyping, type constructors distribute over intersections, including arrows (rule DS-distArrow), records (rule DS-distRcd) and disjoint quantifications (rule DS-distAll).

Another feature of BCD subtyping, which is often overlooked, is the generalization of *top-like types*, i.e. supertypes of Top.

▶ **Definition 3.2** (Specification of top-like types). $\Delta \vdash \rceil A \lceil \triangleq \Delta \vdash A \sim \mathsf{Top}$.

Initially, top-like types include Top and intersections like Top & Top. But the BCD subtyping adds Top → Top to it via rule DS-topArrow, as well as $A \to \mathsf{Top}$ for any type A due to the contravariance of function parameters. Rule DS-topArrow can be viewed as a special case of rule DS-distArrow where intersections are replaced by Top (one can consider it as an intersection of zero components). Like the original F_i^+, we extend this idea to universal types and record types (rules DS-topAll and DS-topRcd).

The most important change is the rule DS-topVar. This rule means that a type variable is top-like if it is disjoint with the bottom type. Every type B is a common supertype of B itself and Bot. If B is disjoint with Bot, then it must be top-like. We proved that subtyping is decidable via an equivalent algorithmic formulation.

The discussion about algorithmic subtyping is in the extended version of the paper.

▶ **Lemma 3.3** (Decidability of subtyping). *$\Delta \vdash A <: B$ is decidable.*

Disjointness. The notion of disjointness (Definition 2.1), defined via subtyping, is used in the original F_i^+, as well as calculi with disjoint intersection types [30]. We proved that our algorithmic definition of disjointness (written as $\Delta \vdash A * B$, in the extended version of the paper) is sound to a specification in terms of top-like types.

▶ **Lemma 3.4** (Disjointness soundness). *If $\Delta \vdash A * B$ then for all type C that $\Delta \vdash A <: C$ and $\Delta \vdash B <: C$ we have $\Delta \vdash \rceil C \lceil$.*

Informally, two disjoint types do not have common supertypes, except for top-like types. This definition is motivated by the desire to prevent ambiguous upcasts on merges. That is, we wish to avoid casts that can extract *different* values of the same type from a merge. Thus in F_i^+ and other calculi with disjoint intersection types, we only allow merges of expressions whose only common supertypes are types that are (equivalent to) the top type. For instance, consider the merge (1 ,, true) ,, (2 ,, ‘c’). The first component of the merge (1 ,, true) has type Int & Bool, while the second component (2 ,, ‘c’) has type Int & Char. This merge is problematic because Int is a supertype of the type of the merge (Int & Bool) & (Int & Char), allowing us to extract two different integers by casting the two terms to Int. Fortunately, our disjointness restriction rejects such merges since the supertype Int is not top-like.

3.3 Bidirectional Typing

The type system of F_i^+ is bidirectional [15], where the subsumption rule is triggered by type annotations. Calculi with a merge operator are incompatible with a general subsumption rule because it cancels disjointness checking. For example, with a general subsumption rule, we can directly use 1 ,, true as a term of type Int since Int & Bool <: Int. Then, merging 1 ,, true with the term false would type-check since disjointness simply checks whether the static types of merging terms are disjoint, and Int is disjoint with Bool. But now, the merge contains two booleans, which would lead to ambiguity if later we wish to extract a boolean value from the merge. The key issue is that a general subsumption rule loses static type information that is necessary to reject ambiguous merges. A bidirectional type system solves this problem by having a more restricted form of subsumption that only works in the checking mode where the type is provided. A more detailed description of the problem for calculi with the merge operator can be found in Huang et al.'s work [22]. We should also remark that this issue of incompatibility with a general subsumption rule is not unique to calculi with a merge operator. It shows up, for instance, in calculi with *gradual typing* [41] and calculi with *record concatenation* and subtyping [9].

Typing modes	$\Leftrightarrow ::= \Leftarrow \mid \Rightarrow$
Pre-values	$u ::= i \mid \top \mid e : A \mid u_1 \,,\, u_2$

$\boxed{\Delta; \Gamma \vdash e \Leftrightarrow A}$ *(Bidirectional Typing)*

$$\textsc{Typ-top}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \mathsf{Top}} \qquad \textsc{Typ-lit}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-var}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad x : A \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A}$$

$$\textsc{Typ-abs}\ \frac{\Delta \vdash B_1 <: A \qquad \Delta; \Gamma, x : A \vdash e \Leftarrow B_2}{\Delta; \Gamma \vdash \lambda x : A.\, e \Leftarrow B_1 \to B_2} \qquad \textsc{Typ-tabs}\ \frac{\Delta \vdash \Gamma \qquad \Delta, X * A; \Gamma \vdash e \Leftarrow B}{\Delta; \Gamma \vdash \Lambda X.\, e \Leftarrow \forall X * A.\, B} \qquad \textsc{Typ-rcd}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A}{\Delta; \Gamma \vdash \{l = e\} \Leftarrow \{l : A\}}$$

$$\textsc{Typ-app}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow A \qquad A \rhd B \to C \qquad \Delta; \Gamma \vdash e_2 \Leftarrow B}{\Delta; \Gamma \vdash e_1\, e_2 \Rightarrow C} \qquad \textsc{Typ-tapp}\ \frac{\Delta; \Gamma \vdash e \Rightarrow B \qquad B \rhd \forall X * C_1.\, C_2 \qquad \Delta \vdash A * C_1}{\Delta; \Gamma \vdash e\, A \Rightarrow C_2[X \mapsto A]} \qquad \textsc{Typ-proj}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \qquad A \rhd \{l : C\}}{\Delta; \Gamma \vdash e.l \Rightarrow C}$$

$$\textsc{Typ-merge}\ \frac{\Delta \vdash A * B \qquad \Delta; \Gamma \vdash e_1 \Rightarrow A \qquad \Delta; \Gamma \vdash e_2 \Rightarrow B}{\Delta; \Gamma \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-mergev}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad u_1 \approx u_2 \qquad \cdot; \cdot \vdash u_1 \Rightarrow A \qquad \cdot; \cdot \vdash u_2 \Rightarrow B}{\Delta; \Gamma \vdash u_1 \,,\, u_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-inter}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A \qquad \Delta; \Gamma \vdash e \Leftarrow B}{\Delta; \Gamma \vdash e \Leftarrow A \,\&\, B}$$

$$\textsc{Typ-fix}\ \frac{\Delta; \Gamma, x : A \vdash e \Leftarrow A}{\Delta; \Gamma \vdash \mathbf{fix}\ x : A.\, e \Rightarrow A} \qquad \textsc{Typ-anno}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A}{\Delta; \Gamma \vdash (e : A) \Rightarrow A} \qquad \textsc{Typ-sub}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \qquad \Delta \vdash A <: B}{\Delta; \Gamma \vdash e \Leftarrow B}$$

Figure 7 Bidirectional typing rules for F_i^+.

Typing. As presented in Figure 7, there are two modes of typing: synthesis ($\Rightarrow$) and checking ($\Leftarrow$). We use $\Leftrightarrow$ as a metavariable for typing modes. $\Delta; \Gamma \vdash e \Leftrightarrow A$ indicates that under type context Δ and term context Γ, the expression e has type A in mode $\Leftrightarrow$. A bidirectional type system directly provides a type-checking algorithm. Δ, Γ, e are all inputs in both modes. Type synthesis generates a *unique* type as the output (also called the inferred type), while type checking takes a type as an input and examines the term.

▶ **Lemma 3.5** (Uniqueness of type synthesis). *If* $\Delta; \Gamma \vdash e \Rightarrow A_1$ *and* $\Delta; \Gamma \vdash e \Rightarrow A_2$ *then* $A_1 = A_2$.

Conversion of typing modes happens in rule Typ-sub. With it, a term with inferred type A can be checked against any B that is a supertype of A. Compared to the original F_i^+, fixpoints are new. They model recursion with a self-reference (x in $\mathbf{fix}\ x : A.\, e$). Other than this, rule Typ-fix is almost the same as rule Typ-anno. It checks the expression e by the annotated type A, with assumption that x has type A in e.

Checking abstractions, type abstractions, and records. To check a function $\lambda x : A.\, e$ against $B_1 \to B_2$ by rule Typ-abs, we track the type of the term variable as *the precise parameter type* A, and check if e can be checked against B_2. B_1 must be a subtype of A

$\boxed{A \rhd B}$ *(Applicative Distribution)*

$$\frac{A_1 \rhd B_1 \to C_1 \qquad A_2 \rhd B_2 \to C_2}{A_1 \,\&\, A_2 \rhd B_1 \,\&\, B_2 \to C_1 \,\&\, C_2}\ \textsc{AD-andArrow}$$

$$\frac{A_1 \rhd \{l : B_1\} \qquad A_2 \rhd \{l : B_2\}}{A_1 \,\&\, A_2 \rhd \{l : B_1 \,\&\, B_2\}}\ \textsc{AD-andRcd}$$

$$\frac{A_1 \rhd \forall X * B_1.\, C_1 \qquad A_2 \rhd \forall X * B_2.\, C_2}{A_1 \,\&\, A_2 \rhd \forall X * B_1 \,\&\, B_2.\, (C_1 \,\&\, C_2)}\ \textsc{AD-andAll}$$

$$\frac{}{A \rhd A}\ \textsc{AD-refl}$$

Figure 8 Applicative distribution rules.

to guarantee the safety of the function application. The type-checking of type abstractions $\Lambda X.\, e$ works by tracking the disjointness relation of the type variable with the context and checking e against the quantified type B. Typing of records works similarly. Additionally, there is a rule Typ-inter, which checks an expression against an intersection type by seperately checking the expression against the composing two types. With this design, we allow $\lambda x : \mathsf{Int}.\, x\, ,,\, \mathsf{true}$ to be checked against $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$.

Application, record projection, and conversion of applicable types. It is not surprising that a merge can act as a function. But in the original F_i^+, this requires annotations since the expression being applied in an application must have an inferred arrow type. Our design, following the λ_i^+ calculus [22], allows a term of an intersection type to directly apply, as long as the intersection type can be converted into an applicable form. For example, $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$ is converted into $(\mathsf{Int} \,\&\, \mathsf{Int}) \to (\mathsf{Int} \,\&\, \mathsf{Bool})$, which is a supertype of the former. When inferring the type of the application $e_1\ e_2$, rule Typ-app first *converts* the inferred type of e_1 into an arrow form $B \to C$ and then checks the argument e_2 against B. If the check succeeds, the whole expression has inferred type C.

In F_i^+, we have three applicable forms: *arrow types*, *record types*, *universal types*. Like rule Typ-app, the typing of type application and record projection also allows the applied term to have an intersection type, and relies on *applicative distribution* to convert the type.

Applicative distribution $A \rhd B$ (defined in Figure 8) takes type A and generates a supertype B that has an applicable form. The first three rules bring all parts of the input intersection type together. For example, assuming that we apply several merged functions whose types are $A_1 \to B_1$, $A_2 \to B_2$, ..., $A_n \to B_n$, the combined function type is $(A_1 \,\&\, ... \,\&\, A_n) \to (B_1 \,\&\, ... \,\&\, B_n)$. It is equivalent to the input type only when A_1, A_2, ..., and A_n are all equivalent. Essentially, applicative distribution ($A \rhd B$) is a subset of subtyping ($A <: B$). The supertype is selected to ensure that when a merge is applied to an argument, every component in the merge is satisfied. Although each one of the three first rules overlaps with the reflexivity rule, for any given type, at most one result has an applicable form.

Since merges are treated as a whole applicable term, programmers can extend functions via a *compositional* approach without modifying the original implementation. It also enables the modular extension of type abstractions and especially records, which play a core role in the trait encoding used in Compositional Programming.

Davies and Pfenning also employ a similar design in their bidirectional type system for refinement intersections [13]. Their type conversion procedure respects subtyping as well. Instead of combining function types, it makes use of $A \,\&\, B <: A$ and $A \,\&\, B <: B$ to enumerate components in intersections and uncover arrows.

Arguments $\quad arg ::= \; e \mid A \mid \{l\}$

Evaluation contexts $\quad E ::= [\,]\; e \mid [\,]\; A \mid [\,].l \mid [\,] \,,,\, v \mid v \,,,\, [\,] \mid [\,] : A$

$\boxed{v \bullet arg \;\hookrightarrow\; u}$ *(Parallel Application)*

PApp-abs
$$\frac{B \rhd C_1 \to C_2 \qquad e_2 \rightsquigarrow_A u}{(\lambda x\!:\!A.\, e_1)\!:\!B \bullet e_2 \;\hookrightarrow\; (e_1[x \mapsto u])\!:\!C_2}$$

PApp-tabs
$$\frac{A \rhd \forall X * B_1.\, B_2}{(\Lambda X.\, e)\!:\!A \bullet C \;\hookrightarrow\; (e[X \mapsto C])\!:\!(B_2[X \mapsto C])}$$

PApp-proj
$$\frac{A \rhd \{l\!:\!B\}}{\{l = e\}\!:\!A \bullet \{l\} \;\hookrightarrow\; e\!:\!B}$$

PApp-merge
$$\frac{v_1 \bullet arg \;\hookrightarrow\; u_1 \qquad v_2 \bullet arg \;\hookrightarrow\; u_2}{v_1 \,,,\, v_2 \bullet arg \;\hookrightarrow\; u_1 \,,,\, u_2}$$

$\boxed{e_1 \;\hookrightarrow\; e_2}$ *(Small-Step Semantics)*

Step-papp
$$\frac{v \bullet e \;\hookrightarrow\; u}{v\; e \;\hookrightarrow\; u}$$

Step-pproj
$$\frac{v \bullet \{l\} \;\hookrightarrow\; u}{v.l \;\hookrightarrow\; u}$$

Step-ptapp
$$\frac{v \bullet A \;\hookrightarrow\; u}{v\; A \;\hookrightarrow\; u}$$

Step-fix
$$\frac{}{\textbf{fix}\; x\!:\!A.\, e \;\hookrightarrow\; e[x \mapsto \textbf{fix}\; x\!:\!A.\, e]\!:\!A}$$

Step-annov
$$\frac{\textsf{pre-value}\; v \qquad v \hookrightarrow_A v'}{v\!:\!A \;\hookrightarrow\; v'}$$

Step-merge
$$\frac{e_1 \;\hookrightarrow\; e_1' \qquad e_2 \;\hookrightarrow\; e_2'}{e_1 \,,,\, e_2 \;\hookrightarrow\; e_1' \,,,\, e_2'}$$

Step-cntx
$$\frac{e \;\hookrightarrow\; e'}{E[e] \;\hookrightarrow\; E[e']}$$

Figure 9 Small-step semantics rules.

Typing merges with disjointness and consistency. Well-typed merges always have inferred types. There are two type synthesis rules for merges, both combining the inferred types of the two parts into an intersection. Typ-merge requires the two subterms to have *disjoint* inferred types, like $1 \,,,\, \mathsf{true}$. Typ-mergev relaxes the disjointness constraint to *consistency* checking (written as $u_1 \approx u_2$) to accept overlapping terms like $1 \,,,\, 1$. Such duplication is meaningless to users but may appear during evaluation. In fact, rule Typ-mergev is designed for metatheory properties, and not to allow more user-written programs [22]. We will state the formal specification of consistency in Section 4.1 and show how it is involved in the proofs of determinism and type soundness. Informally, consistent merges cause no ambiguity in the runtime. For practical reasons, we only consider *pre-values* (defined at the top of Figure 7) in consistency checking, for which the inferred type can be told directly. The algorithms for disjointness and consistency are presented in the extended version of the paper. In general, disjointness and consistency avoid introducing ambiguity of merges, and enable a deterministic semantics for F_i^+.

3.4 Small-Step Operational Semantics

We specify the *call-by-name* reduction of F_i^+ using a small-step operational semantics in Figure 9. Step-papp, Step-pproj, and Step-ptapp are reduction rules for application and record projection. They trigger *parallel application* (defined in the middle of Figure 9) of merged values to the argument. Rule Step-fix substitutes the fixpoint term variable with the fixpoint expression itself. Note that the result is annotated with A. With the explicit type annotation, the result of reduction preserves the type of the original fixpoint expression.

Through rule Step-annov, values are *cast* to their annotated type. Such values must also be pre-values. This is to filter out checkable terms p including bare abstractions or records without annotations, as $p : A$ is a form of value itself and thus should not step.

A merge of multiple terms may reduce in parallel, as shown in rule Step-merge. Only when one side cannot step, the other side steps alone, as suggested by the evaluation context E ,, v and v ,, E. Rule Step-cntx is the reduction rule of expressions within an *evaluation context*. Since the rule can be applied repeatedly, we only need evaluation contexts of depth one (shown at the top of Figure 9). Our operational semantics substitutes arguments *wrapped* by type annotations into function bodies, while it forbids the reduction of records since records are values.

Parallel application. Parallel application is at the heart of what we call *nested composition* in CP. It provides the runtime behavior that is necessary to implement nested composition, and it reflects the subtyping distributivity rules at the term level. A merge of functions is treated as one function. The beta reduction of all functions in a merge happens in *parallel* to keep the consistency of merged terms. For type abstractions or records, things are similar. The parallel application handles these applicable merges uniformly via rule PApp-merge. To align record projection with the other two kinds of application, we define *arguments* which abstract expressions, types, and record labels (at the top of Figure 9). In rule PApp-abs, the argument expression is *wrapped* by the function argument type before we substitute it into the function body. Parallel application of type abstractions substitutes the type argument into the body and annotates the body with the substituted disjoint quantified type. Rule PApp-proj projects record fields. Note these three rules have types to annotate the result, since in Typ-abs, Typ-tabs, and Typ-rcd we only type the expression e inside in *checking* mode. With an explicit type annotation, the application preserves types.

Splittable types. Before explaining wrapping or casting, we first introduce *splittable types* [22], which are a key component of our algorithmic formulations of various relations. Ordinary types are the basic units, values of ordinary types can be constructed without the merge operator. As defined at the top of Figure 10, ordinary types do not have intersection types in positive positions. By contrast, splittable types are isomorphic to appropriate intersections. Recall that in BCD-style distributive rules, arrows distribute over intersection, making $\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}$ equivalent to the intersection $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$. Therefore we say that the former type *splits* into the latter two arrow types. In Figure 10, we extend the type splitting algorithm of λ_i^+ to universal types in correspondence to the distributive subtyping rules (rule DS-distArrow, rule DS-distRcd, and rule DS-distAll). It gives a decision procedure to check whether a type is splittable or ordinary.

▶ **Lemma 3.6** (Type splitting loses no information)**.** *If* $\vdash \Delta$ *and* $\Delta \vdash A$ *and* $B \lhd A \rhd C$ *then* $\Delta \vdash A \sim B \,\&\, C$*.*

Expression wrapping. Rules for expression wrapping ($e \rightsquigarrow_A u$) are listed in the middle of Figure 10. Basically, it splits the type A when possible, annotates a duplication of e by each ordinary part of A, and then composes all of them. The only exception is that it never uses top-like types to annotate terms, to avoid ill-typed results like $\{l = 1\} : \mathsf{Int} \to \mathsf{Top}$, but rather generates a normal value whose inferred type is that top-like type, like $(\lambda x : \mathsf{Int}.\, \top) : \mathsf{Int} \to \mathsf{Top}$ (via the *top-like value generating* function $[\![A^\circ]\!]$, defined in the extended version of the paper).

Ordinary types $\quad A^\circ, B^\circ, C^\circ ::= X \mid \mathsf{Int} \mid \mathsf{Top} \mid \mathsf{Bot} \mid A \to B^\circ \mid \forall X * A.\, B^\circ \mid \{l : A^\circ\}$

$\boxed{B \lhd A \rhd C}$ *(Splittable Types)*

$$\textsc{Sp-arrow}\quad \frac{C_1 \lhd B \rhd C_2}{A \to C_1 \lhd A \to B \rhd A \to C_2} \qquad \textsc{Sp-rcd}\quad \frac{C_1 \lhd B \rhd C_2}{\{l : C_1\} \lhd \{l : B\} \rhd \{l : C_2\}}$$

$$\textsc{Sp-all}\quad \frac{C_1 \lhd B \rhd C_2}{\forall X * A.\, C_1 \lhd \forall X * A.\, B \rhd \forall X * A.\, C_2} \qquad \textsc{Sp-and}\quad \frac{}{A \lhd A \,\&\, B \rhd B}$$

$\boxed{e \leadsto_A u}$ *(Expression Wrapping)*

$$\textsc{EW-top}\quad \frac{\cdot \vdash \rceil A^\circ \lceil}{e \leadsto_{A^\circ} [\![A^\circ]\!]} \qquad \textsc{EW-anno}\quad \frac{\cdot \vdash \neg \rceil B^\circ \lceil}{e \leadsto_{B^\circ} e : B^\circ} \qquad \textsc{EW-and}\quad \frac{B_1 \lhd A \rhd B_2 \qquad e \leadsto_{B_1} u_1 \qquad e \leadsto_{B_2} u_2}{e \leadsto_A u_1 \,,,\, u_2}$$

$\boxed{v_1 \hookrightarrow_A v_2}$ *(Casting)*

$$\textsc{Cast-int}\quad \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{Cast-top}\quad \frac{\cdot \vdash \rceil A^\circ \lceil}{v \hookrightarrow_{A^\circ} [\![A^\circ]\!]} \qquad \textsc{Cast-mergel}\quad \frac{v_1 \hookrightarrow_{A^\circ} v'}{v_1 \,,,\, v_2 \hookrightarrow_{A^\circ} v'} \qquad \textsc{Cast-merger}\quad \frac{v_2 \hookrightarrow_{A^\circ} v'}{v_1 \,,,\, v_2 \hookrightarrow_{A^\circ} v'}$$

$$\textsc{Cast-anno}\quad \frac{\cdot \vdash \neg \rceil B^\circ \lceil \qquad \cdot \vdash A <: B^\circ}{e : A \hookrightarrow_{B^\circ} e : B^\circ} \qquad \textsc{Cast-and}\quad \frac{B_1 \lhd A \rhd B_2 \qquad v \hookrightarrow_{B_1} v_1 \qquad v \hookrightarrow_{B_2} v_2}{v \hookrightarrow_A v_1 \,,,\, v_2}$$

Figure 10 Type splitting, expression wrapping and value casting rules.

Casting. Casting (shown in Figure 10) is the core of the TDOS, and is triggered by the Step-annov rule. Recalling that only values that are also pre-values will be cast, we can always tell the inferred type of the input value and cast it by any supertype of that inferred type. The definition of casting uses the notion of splittable types. In rule Cast-and, the value is cast under two parts of a splittable type separately, and the results are put together by the merge operator. The following example shows that a merge retains its form when cast under equivalent types.

$$\begin{array}{ll} & ((\lambda x : \mathsf{Int}.\, x) : \mathsf{Int} \to \mathsf{Int}) \,,,\, ((\lambda x : \mathsf{Int}.\, \mathsf{true}) : \mathsf{Int} \to \mathsf{Bool}) \\ \hookrightarrow_{(\mathsf{Int}\to\mathsf{Int})\,\&\,(\mathsf{Int}\to\mathsf{Bool})} & ((\lambda x : \mathsf{Int}.\, x) : \mathsf{Int} \to \mathsf{Int}) \,,,\, ((\lambda x : \mathsf{Int}.\, \mathsf{true}) : \mathsf{Int} \to \mathsf{Bool}) \\ \hookrightarrow_{\mathsf{Int}\to\mathsf{Int}\,\&\,\mathsf{Bool}} & ((\lambda x : \mathsf{Int}.\, x) : \mathsf{Int} \to \mathsf{Int}) \,,,\, ((\lambda x : \mathsf{Int}.\, \mathsf{true}) : \mathsf{Int} \to \mathsf{Bool}) \end{array}$$

In the latter case, the requested type is a *function* type, but the result has an intersection type. This change of type causes a major challenge for type preservation.

For ordinary types, rule Cast-int casts an integer to itself under type Int. Under any ordinary top-like type, the cast result is the output of the *top-like value generator*. The casting of values with annotations works by changing the type annotation to the casting (not top-like) supertype. Rule Cast-mergel and rule Cast-merger make a selection between two merged values. The two rules overlap, but for a well-typed value, the casting result is unique.

Example. We show an example to illustrate the behavior of our semantics:

$$\begin{array}{ll}
& \text{Let } f := \lambda x : \mathsf{Int} \,\&\, \mathsf{Top}.\, x \,,, \mathsf{false} \text{ in} \\
& ((f : (\mathsf{Int} \,\&\, \mathsf{Top} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \,\&\, \mathsf{Top} \to \mathsf{Bool})) : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int} \,\&\, \mathsf{Bool})\ (1 \,,, \mathsf{true}) \\
\hookrightarrow^* & \{\text{by rules Step-annov, Cast-and, and Cast-anno}\} \\
& (f : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int}) \,,, (f : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Bool})\ (1 \,,, \mathsf{true}) \\
\hookrightarrow^* & \{\text{by rules Step-papp, EW-and, EW-anno, and EW-top}\} \\
& (((1 \,,, \mathsf{true}) : \mathsf{Int} \,,, \top) \,,, \mathsf{false}) : \mathsf{Int} \,,, (((1 \,,, \mathsf{true}) : \mathsf{Int} \,,, \top) \,,, \mathsf{false}) : \mathsf{Bool} \\
\hookrightarrow^* & \{\text{by rules Step-merge, Step-annov, Cast-int, Cast-mergel, and Cast-merger}\} \\
& 1 \,,, \mathsf{false}
\end{array}$$

This example shows that a function with a splittable type will be cast to a merge of two copies of itself with different type annotations, i.e., two split results. The application of a merge of functions works by distributing the argument to both functions. Finally, casting selects one side of the merge under the annotated type. From this example, we can see that without the precise parameter annotation of a lambda function (here $\mathsf{Int} \,\&\, \mathsf{Top}$), there is no way to filter the argument $1 \,,, \mathsf{true}$, causing a conflict.

4 Type Soundness and Determinism

In this section, we show that the operational semantics of F_i^+ is type-sound and deterministic. In F_i^+, determinism also plays a key role in the proof of type soundness. Proving progress is straightforward and is discussed in the extended version of the paper.

4.1 Determinism

A common problem with determinism for calculi with a merge operator is the ambiguity of selection between merged values. In our system, ambiguity is removed by employing disjointness and consistency constraints on merges via typing.

▶ **Definition 4.1** (Consistency specification). $v_1 \approx_{spec} v_2 \triangleq$ *For all type* A *that* $v_1 \hookrightarrow_A v_1'$ *and* $v_2 \hookrightarrow_A v_2'$ *then* $v_1' = v_2'$.

Two values in a merge have no conflicts as long as casting both values under any type leads to the same result. This specification allows v_1 and v_2 to contain identical expressions (may differ in annotations), and terms with disjoint types as such terms can only be cast under top-like types, and the cast result is only decided by that top-like type.

▶ **Lemma 4.2** (Top-like casting is term irrelevant). *If* $\cdot \vdash \rceil A \lceil$ *and* $v_1 \hookrightarrow_A v_1'$ *and* $v_2 \hookrightarrow_A v_2'$ *then* $v_1' = v_2'$.

This is because casting only happens when the given type is a supertype of the cast value's type, and disjoint types only share top-like types as common supertypes (Lemma 3.4).

▶ **Lemma 4.3** (Upcast only). *If* $\cdot ; \cdot \vdash v \Rightarrow B$ *and* $v \hookrightarrow_A v'$ *then* $\cdot \vdash B <: A$.

With consistency, casting all well-typed values leads to a unique result. The remaining reduction rules, including expression wrapping and parallel application, are trivially deterministic.

▶ **Lemma 4.4** (Determinism of casting). *If* $\cdot ; \cdot \vdash v \Rightarrow B$ *and* $v \hookrightarrow_A v_1$ *and* $v \hookrightarrow_A v_2$*, then* $v_1 = v_2$.

▶ **Theorem 4.5** (Determinism of reduction). *If* $\cdot ; \cdot \vdash e \Rightarrow A$ *and* $e \hookrightarrow e_1$ *and* $e \hookrightarrow e_2$ *then* $e_1 = e_2$.

$\boxed{A \lesssim B}$ *(Isomorphic Subtyping)*

$$\frac{}{A \lesssim A}\ \text{IS-REFL} \qquad \frac{B_1 \lhd B \rhd B_2 \qquad A_1 \lesssim B_1 \qquad A_2 \lesssim B_2}{A_1 \,\&\, A_2 \lesssim B}\ \text{IS-AND}$$

Figure 11 Isomorphic subtyping.

4.2 Preservation

Retaining preservation is challenging. When typing merges, we need to satisfy the extra side conditions in rules Typ-merge and Typ-mergev: disjointness and consistency. While the former only depends on types, the latter needs special care.

Consistency. As discussed in Section 3.4, casting may duplicate terms. For example, $1 \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Int}} 1\,,,\,1$ by rule Cast-and. Rule Typ-mergev is a relaxation of rule Typ-merge to type such merges. We have to ensure any two merged casting results are consistent:

▶ **Lemma 4.6** (Value consistency after casting). *If $\cdot;\cdot \vdash v \Rightarrow C$ and $v \hookrightarrow_A v_1$ and $v \hookrightarrow_B v_2$ then $v_1 \approx v_2$.*

Then we need to make sure that consistency is preserved during reduction.

▶ **Lemma 4.7** (Reduction keeps consistency). *If $\cdot;\cdot \vdash u_1 \Rightarrow A$ and $\cdot;\cdot \vdash u_2 \Rightarrow B$ and $u_1 \approx u_2$ then*

- *if u_1 is a value and $u_2 \hookrightarrow u_2'$ then $u_1 \approx u_2'$;*
- *if u_2 is a value and $u_1 \hookrightarrow u_1'$ then $u_1' \approx u_2$;*
- *if $u_1 \hookrightarrow u_1'$ and $u_2 \hookrightarrow u_2'$ then $u_1' \approx u_2'$.*

Besides, when parallel application substitutes arguments into merges of applicable terms or projects the wished field, consistency is preserved as well. This requirement enforces us to define consistency not only on values but also on pre-values since the application transforms a value merge into a pre-value merge.

▶ **Lemma 4.8** (Parallel application keeps consistency). *If $\cdot;\cdot \vdash v_1 \Rightarrow A$ and $\cdot;\cdot \vdash v_2 \Rightarrow B$ and $v_1 \approx v_2$ and $v_1 \bullet arg \hookrightarrow u_1$ and $v_2 \bullet arg \hookrightarrow u_2$ then $u_1 \approx u_2$ when*

- *arg is a well-typed expression;*
- *or arg is a label;*
- *or arg is a type C; we know $A \rhd \forall X * A_1.\, A_2$ and $B \rhd \forall X * B_1.\, B_2$; and $\cdot \vdash C * A_1 \,\&\, B_1$.*

Our algorithmic formulation of consistency ($u_1 \approx u_2$, presented in the extended version of the paper) keeps the above properties and is sound to the specification (Definition 4.1). The basic idea is to tear all merges apart and compare every component from u_1 and u_2. They are either the same expression with different annotations or have disjoint types.

▶ **Lemma 4.9** (Consistency soundness). *If $v_1 \approx v_2$ then $v_1 \approx_{spec} v_2$.*

Isomorphic subtyping. In F_i^+, types are not always precisely preserved by all reduction steps. Specifically, when we cast a value $v \hookrightarrow_A v'$ (in rule Step-annov) or wrap a term $e \rightsquigarrow_A u$ (in rule PApp-abs), the context expects v' or u to have type A, but this is not always true. In our casting rules shown at the bottom of Figure 10, most values will be reduced to results with the exact type that we want, except for rule Cast-and. The inferred type of the result is always an intersection, which may differ from the original splittable type. To describe the change of types during reduction accurately, we define *isomorphic subtyping* (Figure 11). If $A \lesssim B$, we say A is an isomorphic subtype of B. The following lemma shows that while the two types in an isomorphic subtyping relation may be syntactically different, they are equivalent under an empty type context (i.e. $\cdot \vdash A <: B$ and $\cdot \vdash B <: A$).

▶ **Theorem 4.10** (Isomorphic subtypes are equivalent). *If $A \lesssim B$ then $\cdot \vdash A \sim B$.*

With isomorphic subtyping, we define the preservation property of casting, expression wrapping, and parallel application as follows.

▶ **Lemma 4.11** (Casting preserves typing). *If $\cdot;\cdot \vdash v \Rightarrow A$ and $v \hookrightarrow_B v'$ then there exists a type C such that $\cdot;\cdot \vdash v' \Rightarrow C$ and $C \lesssim B$.*

▶ **Lemma 4.12** (Expression wrapping preserves typing). *If $\cdot;\cdot \vdash e \Leftarrow B$ and $\cdot \vdash B <: A$ and $e \rightsquigarrow_A u$ then there exists a type C such that $\cdot;\cdot \vdash u \Rightarrow C$ and $C \lesssim A$.*

▶ **Lemma 4.13** (Parallel application preserves typing). *If $\cdot;\cdot \vdash v \bullet arg \Rightarrow A$ and $v \bullet arg \hookrightarrow u$ then there exists a type B such that $\cdot;\cdot \vdash u \Rightarrow B$ and $B \lesssim A$.*

Of course, we can prove that the result of casting always has a subtype (or an equivalent type) of the requested type instead of an isomorphic subtype. But it would be insufficient for type preservation of reduction. In summary, if casting or wrapping generates a term of type B when the requested type is A, we need B to satisfy:

- B is a subtype of A because we want a preservation theorem that respects subtyping.
- For any type C, $A * C$ implies $B * C$. This is for the disjointness and consistency checking.
- If A converts into an applicable type C, then B converts into an applicable type too.

Finally, with the lemmas above and isomorphic subtyping, we have the type preservation property of F_i^+. That is, after one or multiple steps of reduction ($\hookrightarrow^*$), the inferred type of the reduced expression is an isomorphic subtype. Therefore, for checked expressions, the initial type-checking always succeeds.

▶ **Theorem 4.14** (Type preservation with isomorphic subtyping). *If $\cdot;\cdot \vdash e \Leftrightarrow A$ and $e \hookrightarrow^* e'$ then there exists a type B such that $\cdot;\cdot \vdash e' \Leftrightarrow B$ and $B \lesssim A$.*

▶ **Corollary 4.15** (Type preservation). *If $\cdot;\cdot \vdash e \Leftrightarrow A$ and $e \hookrightarrow^* e'$ then $\cdot;\cdot \vdash e' \Leftarrow A$.*

5 Related Work

In the following discussion, sometimes we attach the publication year to its calculus name for easy distinction. For instance, F_i^+'19 means the original formulation of F_i^+ by Bi et al. [7].

The merge operator, disjoint intersection types and TDOS. The merge operator for calculi with intersection types was proposed by Reynolds [34]. His original formulation came with significant restrictions to ensure that the semantics is not ambiguous. Castagna [11] showed that a merge operator restricted to functions could model overloading. Dunfield [14]

	$\lambda_{,,}$	λ_i'16	F_i	λ_i^+'18	F_i^+'19	λ_i	λ_i^+	F_i^+
Disjointness	○	●	●	●	●	●	●	●
Unrestricted Intersections	●	○	○	●	●	●	●	●
Determinism / Coherence	No	Coh.	Coh.	Coh.	Coh.	Det.	Det.	Det.
Recursion	●	○	○	○	○	●	●	●
Direct Semantics	●	○	○	○	○	●	●	●
Subject Reduction	○	-	-	-	-	●	●	●
Distributive Subtyping	○	○	○	●	●	○	●	●
Disjoint Polymorphism	○	○	●	○	●	○	○	●
Evaluation Strategy	CBV	CBV	CBV	CBV	CBV	CBV	CBV	CBN

Figure 12 Summary of intersection calculi with the merge operator.
(● = yes, ○ = no, - = not applicable).

proposed a calculus, which we refer to as $\lambda_{,,}$, with an unrestricted merge operator. While powerful, $\lambda_{,,}$ lacked both determinism and subject reduction, though type safety was proved via a *type-directed* elaboration semantics.

To address the ambiguity problems in Dunfield's calculus, Oliveira et al. [30] proposed λ_i'16, which only allows intersections of disjoint types. With that restriction and the use of an elaboration semantics, it was then possible to prove the coherence of λ_i'16, showing that the semantics was not ambiguous. Bi et al. [6] relaxed the disjointness restriction, requiring it only on merges, in a new calculus called λ_i^+'18 (or NeColus). This enabled the use of unrestricted intersections in λ_i^+'18. In addition, they added a more powerful subtyping relation based on the well-known BCD subtyping [4] relation. The new subtyping relation, in turn, enabled nested composition, which is a fundamental feature of Compositional Programming. Unfortunately, both unrestricted intersections and BCD subtyping greatly complicated the coherence proof of λ_i^+'18. To address those issues, Bi et al. turned to an approach based on logical relations and a notion of contextual equivalence.

To address the increasing complexities arising from the elaboration semantics and the coherence proofs, Huang et al. [20, 22] proposed a new approach to model the type-directed semantics of calculi with a merge operator. The type-directed elaboration in λ_i'16 and λ_i^+'18 is replaced by a direct *type-directed operational semantics* (TDOS). In the new TDOS formulations of λ_i and λ_i^+, coercive subtyping is removed since subtyping no longer needs to generate explicit coercion for the elaboration to a target calculus. Instead, runtime implicit (up)casting is used. This is implemented by the casting relation, which was originally called *typed reduction*. Our work adopts TDOS and adds disjoint polymorphism. Disjoint polymorphism is used in Compositional Programming to enable techniques such as polymorphic contexts. We also change the evaluation strategy from call-by-value (CBV) to call-by-name (CBN), motivated by the elaboration of trait instantiation in Compositional Programming. Otherwise, with a CBV semantics, many uses of trait instantiation would diverge.

Calculi with disjoint polymorphism. Disjoint polymorphism was originally introduced in a calculus called F_i by Alpuim et al. [2]. A disjointness constraint is added to universal quantification in order to allow merging components whose type contains type variables. Later, Bi et al. [7] augment it with distributive subtyping in the F_i^+'19 calculus. In addition, the bottom type is added, and unrestricted intersection types are also allowed to fully encode row and bounded polymorphism [44]. Compared to F_i^+'19, our new formulation of F_i^+ adopts a direct semantics, based on a TDOS approach, where simpler proofs of determinism supersede the original proofs of coherence. As a result, recursion and impredicative polymorphism can be easily added. Both features are important to fully support the trait encoding in Compositional Programming. A detailed comparison of calculi with a merge operator, which summarizes our discussion on related work, can be found in Figure 12.

F_i^+ versus $F_{<:}$. There are quite a few typed object encodings in the literature [8], most of which are based on $F_{<:}$ [10]. As it is not our goal in this paper to encode full OOP in F_i^+, we will not compare our trait encoding with other object encodings. However, it is still interesting to compare F_i^+ with $F_{<:}$. Some disadvantages of $F_{<:}$ have been studied in the literature. It has been shown that, with bounded quantification, the subtyping of $F_{<:}$ is undecidable [32], and some useful operations like polymorphic record updates [9] are not directly supported. F_i^+ does not have these drawbacks. F_i^+ has decidable subtyping. For $F_{<:}$ the most common decidable fragment is the so-called kernel $F_{<:}$ variant [10]. Xie et al. [44] have shown that kernel $F_{<:}$ is encodable in F_i^+. Therefore the bounded quantification that is present in kernel $F_{<:}$ can be expressed in F_i^+ as well. In addition, polymorphic record updates can be easily encoded without extra language constructs. For example, concerning a polymorphic record that contains an `x` field among others (`rcd : { x: Int } & R`), the record update `{ rcd with x = 1 }` can be encoded in F_i^+ as `{ x = 1 } ,, (rcd : R)`. In other words, we can rewrite whichever fields we want and then merge the remaining polymorphic part back.

F_i^+ versus row-polymorphic calculi. Row polymorphism provides an alternative way to model extensible record types in System F-like calculi. There are many variants of row-polymorphic calculi in the literature [9, 19, 24, 38]. Among them, the most relevant one with respect to our work is $\lambda^{||}$ by Harper and Pierce [19]. Disjoint quantification has a striking similarity to *constrained quantification* in $\lambda^{||}$. Their *compatibility* constraint plays a similar role to *disjointness* in our system. Furthermore, their merge operator (`||`) can concatenate either two records like our merge operator (, ,) or two record types like our intersection type operator (&). However, their compatibility constraint and merge operator are only applicable to record types, while we generalize them to arbitrary types. $\lambda^{||}$ has no subtyping and does not allow for distributivity and nested composition either. Disjoint polymorphism also subsumes the form of row polymorphism present in $\lambda^{||}$ as demonstrated by Xie et al. [44]. We refer to Xie et al.'s work for an extended discussion of the relationship between F_i^+ and various other row polymorphic calculi.

Semantics for type-dependent languages. The elaboration semantics approach is commonly used to model the semantics of type-dependent languages and calculi. The appeals of the elaboration semantics are simple type-safety proofs, and the fact that they directly offer an implementation technique over conventional languages without a type-dependent semantics. For instance, the semantics of type-dependent languages with *type classes* [18, 43], *Scala-style implicits* [27, 29] or *gradual typing* [40] all use an elaboration semantics. In contrast, in the past, more conventional direct formulations using an operational semantics have been avoided for languages with a type-dependent semantics. A problem is that the type-dependent semantics introduces complexity in the formulation of an operational semantics since enough type information should be present at runtime and type information needs to be properly propagated. Early work on the semantics of type classes [23, 28], for instance, attempted to employ an operational semantics. However, those approaches had significant practical restrictions in comparison to conventional type classes. The TDOS approach has shown how to overcome important issues when modeling the direct semantics of type-dependent languages. An important advantage of the TDOS approach is that it removes the need for non-trivial coherence proofs. The TDOS approach has also been recently shown to work for modeling the semantics of gradually typed languages directly [45].

6 Conclusion

In this paper, we presented a new formulation of the F_i^+ calculus and showed how it serves as a direct foundation for Compositional Programming. In contrast to the original F_i^+, we adopt a direct semantics based on the TDOS approach and embrace call-by-name evaluation. As a result, the metatheory of F_i^+ is significantly simplified, especially due to the fact that a coherence proof based on logical relations and contextual equivalence is not needed. In addition, our formulation of F_i^+ enables recursion and impredicative polymorphism, validating the original trait encoding by Zhang et al. [46]. We proved the type-soundness and determinism of F_i^+ using the Coq proof assistant. Our research explores further possibilities of the TDOS approach and shows some novel notions that could inspire the design of other calculi with similar features.

Although F_i^+ is already expressive enough to work as a core calculus of the CP language, some useful constructs like type operators are missing. We leave the extension of type-level operations for future work. Another interesting design choice that we want to explore is to lazily evaluate both sides of merges, just like what we have done for record fields, which can help avoid some redundant computation on the unused side of a merge.

References

1. Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, 2006.
2. João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.
3. Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.
4. Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.
5. Xuan Bi and Bruno C. d. S. Oliveira. Typed First-Class Traits. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.
6. Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.
7. Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In *European Symposium on Programming (ESOP)*, 2019.
8. Kim B Bruce, Luca Cardelli, and Benjamin C Pierce. Comparing object encodings. In *International Symposium on Theoretical Aspects of Computer Software*, pages 415–438. Springer, 1997.
9. Luca Cardelli and John C Mitchell. Operations on records. In *International Conference on Mathematical Foundations of Programming Semantics*, 1989.
10. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.
11. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Conference on LISP and Functional Programming*, 1992.
12. William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
13. Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *International Conference on Functional Programming (ICFP)*, 2000.
14. Jana Dunfield. Elaborating intersection and union types. *Journal of Functional Programming (JFP)*, 24(2-3):133–165, 2014.

15 Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021. doi:10.1145/3450952.

16 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

17 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

18 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996.

19 Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Principles of Programming Languages (POPL)*, 1991.

20 Xuejing Huang and Bruno C. d. S. Oliveira. A type-directed operational semantics for a calculus with a merge operator. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:32, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.26.

21 Xuejing Huang and Bruno C d S Oliveira. Distributing intersection and union types with splits and duality (functional pearl). *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–24, 2021.

22 Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. Taming the merge operator. *Journal of Functional Programming*, 31:e28, 2021. doi:10.1017/S0956796821000186.

23 Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *ESOP '88*, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

24 Daan Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005.

25 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 333–343, 1995.

26 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. In *OOPSLA*, 2018.

27 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: Foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

28 Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 135–146, New York, NY, USA, 1995. Association for Computing Machinery.

29 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360. ACM, 2010. doi:10.1145/1869459.1869489.

30 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.

31 Benjamin C Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical report, Carnegie Mellon University, 1989.

32 Benjamin C Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.

33 John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

34 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.

35 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.

36 John C Reynolds. Design of the programming language forsythe. In *ALGOL-like languages*, pages 173–233. Birkhauser Boston Inc., 1997.

37 Tom Schrijvers and Bruno C. d. S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN international conference on functional programming*, pages 32–44, 2011.

38 Mark Shields and Erik Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 261–275, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.360230.

39 Jeremy G. Siek. Transitivity of subtyping for intersection types. *CoRR*, abs / 1906.09709, 2019. arXiv:1906.09709.

40 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

41 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2007.

42 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

43 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

44 Ningning Xie, Bruno C d S Oliveira, Xuan Bi, and Tom Schrijvers. Row and bounded polymorphism via disjoint polymorphism. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

45 Wenjia Ye, Bruno C. d. S. Oliveira, and Xuejing Huang. Type-directed operational semantics for gradual typing. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 12:1–12:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

46 Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. Compositional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(3):1–61, 2021.

Low-Level Bi-Abduction

Lukáš Holík
FIT, Brno University of Technology, Czech Republic

Petr Peringer
FIT, Brno University of Technology, Czech Republic

Adam Rogalewicz
FIT, Brno University of Technology, Czech Republic

Veronika Šoková
FIT, Brno University of Technology, Czech Republic

Tomáš Vojnar
FIT, Brno University of Technology, Czech Republic

Florian Zuleger
Faculty of Informatics, TU Wien, Austria

Abstract

The paper proposes a new static analysis designed to handle open programs, i.e., fragments of programs, with dynamic pointer-linked data structures – in particular, various kinds of lists – that employ advanced low-level pointer operations. The goal is to allow such programs be analysed without a need of writing analysis harnesses that would first initialise the structures being handled. The approach builds on a special flavour of separation logic and the approach of bi-abduction. The code of interest is analyzed along the call tree, starting from its leaves, with each function analysed just once without any call context, leading to a set of contracts summarizing the behaviour of the analysed functions. In order to handle the considered programs, methods of abduction existing in the literature are significantly modified and extended in the paper. The proposed approach has been implemented in a tool prototype and successfully evaluated on not large but complex programs.

2012 ACM Subject Classification Theory of computation → Separation logic; Theory of computation → Logic and verification; Software and its engineering → Formal software verification

Keywords and phrases programs with dynamic linked data structures, programs with pointers, low-level pointer operations, static analysis, shape analysis, separation logic, bi-abduction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.19

Related Version *Extended Version*: arXiv:2205.02590 [22]

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.11

Funding The Czech authors were supported by the project 20-07487S of the Czech Science Foundation, the FIT BUT internal project FIT-S-20-6427, and L. Holík by the ERC.CZ project LL1908.

1 Introduction

Programs with complex dynamic data structures and pointer operations are notoriously difficult to write and understand. This holds twice when a need to achieve the best possible performance drives programmers, especially those working in the C language on which we concentrate, to start using advanced low-level pointer operations such as pointer arithmetic, bit-masking information on pointers, address alignment, block operations with blocks that are split to differently sized fields (of size not known in advance), which can then be merged again, and reinterpreted differently, and so on. It may then easily happen that the resulting programs

© Lukáš Holík, Petr Peringer, Adam Rogalewicz, Veronika Šoková, Tomáš Vojnar, and Florian Zuleger; licensed under Creative Commons License CC-BY 4.0

contain nasty errors, such as null-pointer dereferences, out-of-bound references, double free operations, or memory leaks, which can manifest only under some rare circumstances, may escape traditional testing, and be difficult to discover once the program is in production.

To help discover such problems (or show their absence), suitable static analyses with formal roots may help. However, the problem of analysing programs with dynamic pointer-linked data structures, sometimes referred to as *shape analysis*, belongs among the most difficult analysis problems, which is related to a need of efficiently encoding and handling potentially infinite sets of graph structures of in-advance unknown shape and unbounded size, corresponding to the possible memory configurations.

Moreover, the problem becomes even harder when one needs to analyse not entire programs, equipped with some analysis harness generating instances of the data structures to be handled, but just *fragments of code*, which simply start handling some dynamic data structures through pointers without the structures being initialised first. At the same time, in practice, the possibility of analysing code fragments is highly preferred since programmers do not like writing specialised analysis harnesses for initialising data structures of the code to be analysed (not speaking about that writing such harnesses is error-prone too). Moreover, the possibility of analysing code fragments can also help scalability of the analysis since it can then be performed in a modular way.

In this paper, we propose a new analysis designed to analyse programs and even *fragments* of programs with *dynamic pointer-linked data structures* that can use advanced *low-level pointer-manipulating operations* of the form mentioned above. In particular, we concentrate on sequential C programs without recursion and without function pointers manipulating various forms of *lists* – singly-linked, doubly-linked, circular, nested, and/or intrusive, which are perhaps the most common kind of dynamic linked data structures in practice.

Our approach uses a special flavor of *separation logic* (SL) [33, 24] with *inductive list predicates* [2] to characterize sets of program configurations. To be able to handle code fragments, we adopt the principle of *bi-abductive analysis* proposed over SL for analysing programs without low-level pointer operations in [6, 7]. Our work can thus be viewed as an extension of the approach of [6, 7] to programs with truly low-level operations (i.e., pointer arithmetic, bit-masking on pointers, block operations with blocks of variable size, their splitting to fields of in-advance-not-fixed size, merging such fields back, and reinterpreting them differently, etc.). As will become clear, handling such programs requires rather non-trivial changes to the abduction procedure used in [6, 7] – intuitively, one needs new analysis rules for block splitting and merging, new support for operations such as pointer plus, pointer minus, or block operations (like `memcpy`), and also modified support for operations like memory allocation or deallocation (to avoid deallocation of parts of blocks). Moreover, to support splitting of memory blocks to parts, gradually learning their bounds and fields, and to allow for embedding data structures into other data structures not known in advance (as commonly done, e.g., in the so-called intrusive lists), we even switch from using the traditional *per-object separating conjunction* in our SL to a *per-field separating conjunction* (as used, e.g., in [14] in the context of analysing so-called overlaid data structures), requiring separation not on the level of allocated memory blocks but their fields. As an additional benefit, our usage of per-field separating conjunction then allows us to represent more compactly even some operations on traditional data structures (without low-level pointer manipulation).

As common in bi-abductive analyses, we analyse programs, or their fragments, along their *call tree*, starting from the *leaves* of the call tree (for the time being, we assume working with non-recursive programs only). Each function is analysed just once, without any knowledge about its possible call contexts. For each function, the analysis derives a set of so-called

contracts, which can then be used when this function is called from some other function higher up in the call hierarchy. A contract for a function f is a pair (P, Q) where P is a precondition under which f can be safely executed (without a risk of running into some memory error such as a null-dereference), and Q is a postcondition that is guaranteed to be satisfied upon exit from f provided it was called under the given precondition. Both P and Q are described using our flavor of SL. In fact, as also done in [6, 7], our analysis runs in two phases: the first phase derives the preconditions, while the second phase computes the postconditions. Like in [6, 7], the computed set of contracts may *under-approximate* the set of all possible safe preconditions of f (e.g., some extreme but still safe preconditions need not be discovered). However, for each computed contract (P, Q), the post-condition Q is guaranteed to *over-approximate* all configurations that result from calling the function under the pre-condition P.

We have implemented our approach in a prototype tool called *Broom*. We have applied the tool to a selection of code fragments dealing with various kinds of lists, including very advanced implementations taken from the Linux kernel as well as the intrusive list library (for a reference, see our experimental section). Although the code is not large in the number of lines of code, it contains very advanced pointer operations, and, to the best of our knowledge, Broom is currently the only analyser that is capable of analysing many of the involved functions.

Related work

In the past (at least) 25 years there have appeared numerous approaches to automated *shape analysis* or, more generally, analysis of programs with unbounded dynamically-linked data structures. These approaches differ in the formalisms used for encoding sets of configurations of programs with such data structures, in their level of automation, classes of supported data structures, and/or properties of programs that are targeted by the analysis: see, e.g., [25, 34, 2, 37, 9, 39, 38, 20, 10, 3, 16, 21, 31].

Not many of the existing approaches offer a reasonably general support of *low-level pointer operations* (such as pointer arithmetic, address alignment, masking information on pointers, block operations, etc.). Some support of low-level pointer operations appears in multiple of these approaches, but it is often not much documented. In fact, such a support often appears in some *ad hoc* extension of the tool implementing the given approach only, without any description whatsoever. According to the best of our knowledge, the approach of [16], based on so-called *symbolic memory graphs (SMGs)*, currently provides probably the most systematic and generic solution for the case of programs with low-level pointer operations and various kinds of linked lists (including advanced list implementations such as those used in the Linux kernel). Specialised approaches to certain classes of low-level programs, namely, *memory allocators*, then appear, e.g., in [5, 19].

In this work, we get inspired by some of the analysis capabilities of [16], but we aim at removing one of its main limitations – namely, the fact that it cannot be applied to a *fragment of code*. Indeed, [16] expects the analysed program to be *closed*, i.e., the analysed functions must be complemented by a *harness* that initializes all the involved data structures, which severely limits applicability of the approach in practice (since programmers are often reluctant to write specialised analysis harnesses).

Approaches allowing one to analyse *open code*, i.e., *code fragments*, with dynamic linked data structures are not frequent in the literature. Perhaps the best known of these works is the approach of *bi-abduction* based on *separation logic* with (possibly nested) list predicates

proposed in [6, 7] and currently available in the Infer analyser [4].[1] This approach is another of the approaches that inspired our work, and we will be referring to various technical details of that paper later on. However, despite Infer contains some support of pointer arithmetic, it is not very complete (as our experiments will show), and the approach presented in [6, 7] does not at all study low-level pointer operations of the form that we aim at in this paper. Moreover, it turns out that adding a support of such operations (e.g., dealing with blocks of memory of possibly variable size, splitting them to fields of variable size, merging such fields back and reinterpreting their contents differently, having pointers with variable offsets, supporting rich pointer arithmetic, etc.) requires rather non-trivial changes and extensions to the bi-abduction mechanisms used in [6, 7].

An approach of *second-order bi-abduction* based also on separation logic was proposed in [28] and several follow-up papers such as [11]. The authors consider recursive programs with pointers and propose a calculus for automatic derivation of sets of equations describing the behaviour of particular functions. A solution of such a set of equations leads to a set of contracts for the considered functions. The technique is in some sense quite general – unlike [6, 7] and unlike our approach, it can even automatically learn *recursive predicates* describing the involved data structures, including trees, skip lists, etc. Moreover, the derivation of the equations is a cheap procedure, and no widening is needed, again unlike in [6, 7] and unlike in our approach. On the other hand, finding a solution of the generated equations is a hard problem, and the authors provide a simple heuristic designed for a specific shape of the equations only, which fails in various other cases.

Finally, we mention the Gillian project, a *language-independent framework* based on separation logic for the development of *compositional symbolic analysis* tools, including tools for whole-program symbolic execution, verification of annotated code, as well as bi-abduction [36, 35, 30, 29]. The works on Gillian concentrate on the generic framework it develops, and the published description of the supported bi-abductive analysis, perhaps most discussed in [35], is unfortunately not very detailed. In particular, it is not clear whether and how much the approach supports the low-level features of pointer manipulation that we are aiming at here (e.g., pointer arithmetic, bit-masking on addresses, etc.). According to the source code that we were able to find in the Gillian repository, the examples mentioned in the part of [35] devoted to bi-abduction do not use low-level pointer manipulation features such as pointer arithmetic. It is also mentioned in [35] that Gillian supports bi-abduction up to a predefined bound only, whereas we do not require such a bound. Further, in contrast to the present work, [35] assumes that the size of memory chunks being dynamically allocated is known, and the complex reasoning needed to resolve this issue is left for the future.

We also note that there is a vast body of work on *automated decision procedures* for various fragments of separation logic and problems such as satisfiability and entailment – see, e.g., [18, 23, 26, 27, 17]. However, it is not immediate how to apply these logics inside a program analysis tool. This is because the best (i.e., logically weakest) solution to the abduction problem $\varphi * [?] \models \psi$, which is a central problem for compositional program analyses, with $*$ being the separating conjunction, is given by the formula $\varphi \mathrel{-\!\!*} \psi$, which makes use of the magic wand operator $\mathrel{-\!\!*}$, and the cited logics do not provide support for the magic wand. This is for principle reasons: it has been observed in the literature that magic wand operators are "difficult to eliminate" [1]; further, it has been shown that adding only the

[1] The approach [6, 7] mentions a generalisation to other classes of data structures, but – to the best of our knowledge – this extension has not been implemented and evaluated, and so it is not clear how well it would work in practice.

singly-linked list-segment predicate to a propositional separation logic that includes the magic wand already leads to undecidability of the satisfiability problem [13]. A notable exception is the recent work [32] on a new semantics for separation logic, which enables decidability of a propositional separation logic that includes the magic wand and the singly-linked list-segment predicate (and also discusses applications to the abduction problem); however, the fragment considered in [32] is not expressive enough to cover the low-level features considered in this work such as, pointer arithmetic, memory blocks, etc., and, at present, it is unclear whether the decidability result can be extended to a richer logic. For the above reasons, we will in this paper not target a complete procedure for the (bi-)abduction problem, but rather, following [6, 7], develop approximate procedures and evaluate their usefulness in our case studies.

Main contributions of the paper

The paper proposes a new approach for automated bi-abductive analysis of programs and fragments of programs with pointers, different kinds of linked lists, and low-level memory operations. The approach is formalised, implemented in a prototype tool, and experimentally evaluated. In summary, we make the following contributions:

- A specialised dialect of separation logic suitable for automated abductive analysis of programs with lists and low-level memory operations (we use a separating conjunction between single fields and not whole memory blocks as in related approaches, and support fields of unknown and even variable size as well as unknown block boundaries).
- Contracts for basic programming statements that reflect our low-level memory model (see, e.g., the contracts of the `malloc` and `free` statements), and support for specific statements that permit low-level pointer manipulation (e.g., pointer addition).
- A set of rules for automated abductive analysis, which not only includes variants of rules from related approaches, but also new kinds of rules required for handling low-level memory operations (e.g., block splitting).
- A prototype implementation that supports bit-precise reasoning based on a reduction of (un-)satisfiability of separation logic to (un-)satisfiability of SMT over the bit-vectors.
- An experimental evaluation of the approach on a number of challenging programs.

2 An Illustration of the Approach on an Example

Before we start with a systematic description of our approach, we present its core ideas on an example. We attempt to informally explain the involved notions, yet, due to the complexity of the issues, some prior knowledge of separation logic with *inductive list predicates*, e.g., [2], and ideally also bi-abduction analysis [6, 7] is helpful.

As our illustrative example, we consider the code manipulating cyclic doubly-linked lists shown in Fig. 1.[2] The example is inspired by the principle of *intrusive lists* (as used, e.g., in Linux kernel lists) where all list operations are defined on some simple list-linking structure that is then nested into user-defined structures. It is these user-defined structures that carry the data actually stored in the lists. The list manipulating functions, however, know nothing about these larger structures. However, the fact that contracts (summaries) derived for

[2] The code is written in C. Our later presented low-level programming language for which we will formalise our approach is not C but rather close to some of the intermediate languages used when compiling C. We, however, feel that describing the example in such a language would not be very understandable. Moreover, all constructions used in our example can be translated to the later considered language.

functions dealing with the small linking structures are later to be applied on the larger, user-defined structures is already problematic for some existing analyses.

In the code of our illustrative example, the function `init_dll` creates an initial cyclic doubly-linked list consisting of a single node. The function `insert_after` can then insert a new element into the list after its item pointed by l.

Let us note that while the code of the example in Fig. 1 may seem to not use pointer arithmetic, the code in fact uses pointer arithmetic on the level of the intermediate code we analyse. Indeed, each expression `x->field` is translated to `*(x+offsetof(field))`. It is of course true that once all the types and fields are known and fixed, one can avoid dealing with pointer arithmetic in this case. On the other hand, the fact that we systematically handle it through pointer arithmetic allows us to smoothly handle even the cases when the types and offsets stop being known and/or constant (upon which approaches based on dealing with field names fail).

As indicated already in the introduction, we analyse the given code fragment according to its *call tree*, starting from the leaves (assuming there is no recursion). Each function is analysed just once, without any call context. If successful, the analysis derives a set of contracts for the given function where each contract is a pair (P, Q) consisting of a (conjunctive) pre-condition and (a possibly disjunctive) post-condition. In our introductory example, we will restrict ourselves to the simplest case, namely, having a single, purely conjunctive contract. In the contracts, both the pre- and post-condition are expressed as SL formulae. The analysis is *compositional* in that contracts derived for some functions are then used when analysing functions higher up in the call hierarchy (moreover, we will view even particular pointer manipulating statements as special atomic functions and describe them by pre-defined contracts).

We begin the illustration of our analysis by analysing the `init_dll` function. We start the analysis by annotating the first line by the pair $(x = X, x = X)$. In this pair, the first component is the so-far derived pre-condition of the function, and the second component is the current symbolic state of the function under analysis. Here, the variable X records the value of the program variable x at the beginning of the function. While x will be changing in the function, X will never change, and we will be able to gradually generate constraints on its value to express what must hold for x at the entry of the function.

After symbolically executing the statement `x->next = x`, we derive that the address X must correspond to some allocated memory, containing some unknown value L_1. This gives us the pre-condition $X \mapsto L_1$ that is an SL formula stating exactly the fact that X is allocated and stores the value L_1. The symbolic state is then advanced to say that X is allocated and stores the value X, i.e., it points to itself, which is encoded as $X \mapsto X$ in SL.

After the subsequent statement `x->prev = x`, assuming that we work with 64 bit (i.e., 8 bytes) wide addresses, we add to the precondition the fact that the memory address $X + 8$ is allocated as well. Moreover, the formula $\mathfrak{b}(X) = \mathfrak{b}(X + 8)$ says that X and $X + 8$ belong to the same memory block, i.e., they were, e.g., allocated using one `malloc` statement (in fact, we use $\mathfrak{b}(X)$ to denote the – so-far unknown – base address of the block). The symbolic state is updated by the fact that the value at the address $X + 8$ is also equal to X, i.e., $X + 8 \mapsto X$.

Since there are no further statements in the function, there is no branching, no loops, and all the statements are deterministic, the final *contract* for the function is unique and consists of the final pre-condition $P \equiv X \mapsto L_1 * X + 8 \mapsto L_2 * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X$ and the post-condition $Q \equiv X \mapsto X * X + 8 \mapsto X * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X$ obtained from the final symbolic state. Here, we use "$*$" to denote a *per-field separating conjunction*, which,

struct dll { struct dll *next, *prev; };
struct emb_dll {int value; struct dll link; };

void **init_dll**(struct dll *x) {

$P \equiv x = X, \quad Q \equiv x = X$

x–¿next = x;

$P \equiv X \mapsto L_1 * x = X, \quad Q \equiv X \mapsto X * x = X$

x–¿prev = x;

$P \equiv X \mapsto L_1 * X + 8 \mapsto L_2 * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X,$
$Q \equiv X \mapsto X * X + 8 \mapsto X * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X$

} summary:

$P \equiv X \mapsto L_1 * X + 8 \mapsto L_2 * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X,$
$Q \equiv X \mapsto X * X + 8 \mapsto X * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X$

void **insert_after**(struct dll *l, *j) {

$P \equiv l = L * j = J, \quad Q \equiv l = L * j = J$

struct dll *n = l–¿next;

$P \equiv L \mapsto N * l = L * j = J, \quad Q \equiv L \mapsto N * l = L * j = J * n = N$

j–¿next = n;

$P \equiv L \mapsto N * J \mapsto B_1 * l = L * j = J, \quad Q \equiv L \mapsto N * J \mapsto N * l = L * j = J * n = N$

j–¿prev = l;

$P \equiv L \mapsto N * J \mapsto B_1 * J + 8 \mapsto B_2 * \mathfrak{b}(J) = \mathfrak{b}(J + 8) * l = L * j = J,$
$Q \equiv L \mapsto N * J \mapsto N * J + 8 \mapsto L * \mathfrak{b}(J) = \mathfrak{b}(J + 8) * l = L * j = J * n = N$

l–¿next = j;

$P \equiv L \mapsto N * J \mapsto B_1 * J + 8 \mapsto B_2 * \mathfrak{b}(J) = \mathfrak{b}(J + 8) * l = L * j = J,$
$Q \equiv L \mapsto J * J \mapsto N * J + 8 \mapsto L * \mathfrak{b}(J) = \mathfrak{b}(J + 8) * l = L * j = J * n = N$

n–¿prev = j;

$P \equiv L \mapsto N * J \mapsto B_1 * J+8 \mapsto B_2 * N+8 \mapsto B_3 * \mathfrak{b}(J) = \mathfrak{b}(J+8) * \mathfrak{b}(N) = \mathfrak{b}(N+8) * l = L * j = J,$
$Q \equiv L \mapsto J * J \mapsto N * J + 8 \mapsto L * N + 8 \mapsto J * \mathfrak{b}(J) = \mathfrak{b}(J + 8) * \mathfrak{b}(N) = \mathfrak{b}(N + 8) * l = L * j = J * n = N$

} summary:

$P \equiv L \mapsto N * J \mapsto B_1 * J + 8 \mapsto B_2 * N + 8 \mapsto B_3 * \mathfrak{b}(J) = \mathfrak{b}(J + 8) * \mathfrak{b}(N) = \mathfrak{b}(N + 8) * l = L * j = J,$
$Q \equiv L \mapsto J * J \mapsto N * J + 8 \mapsto L * N + 8 \mapsto J * \mathfrak{b}(J) = \mathfrak{b}(J + 8) * \mathfrak{b}(N) = \mathfrak{b}(N + 8) * l = L * j = J$

int **main**() {

$P \equiv \mathsf{emp}, \quad Q \equiv \mathsf{emp}$

struct emb_dll *x = malloc(sizeof(struct emb_dll));

$P \equiv \mathsf{emp}, \quad Q \equiv \exists X.\ X \mapsto \top[24] * X = \mathfrak{b}(X) * x = X$

init_dll(&(x–¿link));

$P \equiv \mathsf{emp}, \quad Q \equiv \exists X, L_1.\ X \mapsto \top[8] * L_1 \mapsto L_1 * L_1 + 8 \mapsto L_1 * X = \mathfrak{b}(X) = \mathfrak{b}(L_1) = \mathfrak{b}(L_1 + 8) * L_1 = X + 8 * x = X$

struct emb_dll *i = malloc(sizeof(struct emb_dll));

$P \equiv \mathsf{emp}, \quad Q \equiv \exists I, X, L_1.\ I \mapsto \top[24] * X \mapsto \top[8] * L_1 \mapsto L_1 * L_1 + 8 \mapsto L_1 * L_1 = X + 8 * X = \mathfrak{b}(X) = \mathfrak{b}(L_1) = \mathfrak{b}(L_1 + 8) * I = \mathfrak{b}(I) * x = X * i = I$

init_dll(&(i–¿link));

$P \equiv \mathsf{emp}, \quad Q \equiv \exists I, X, L_1, L_2.\ i \mapsto \top[8] * L_2 \mapsto L_2 * L_2 + 8 \mapsto L_2 * X \to \top[8] * L_1 \mapsto L_1 * L_1 + 8 \mapsto L_1 * L_2 = I + 8 * L_1 = X + 8 * X = \mathfrak{b}(X) = \mathfrak{b}(L_1) = \mathfrak{b}(L_1 + 8) * I = b(I) = b(L_2) = b(L_2 + 8) * x = X * i = I$

insert_after(&(x–¿link), &(i–¿link));

$P \equiv \mathsf{emp}, \quad Q \equiv \exists I, X, L_1, L_2.\ I \mapsto \top[8] * L_2 \mapsto L_1 * L_2 + 8 \mapsto L_1 * X \mapsto \top[8] * L_1 \mapsto L_2 * L_1 + 8 \mapsto L_2 * L_2 = I + 8 * L_1 = X + 8 * X = \mathfrak{b}(X) = b(L_1) = \mathfrak{b}(L_1 + 8) * I = \mathfrak{b}(I) = \mathfrak{b}(L_2) = \mathfrak{b}(L_2 + 8) * x = X * i = I$

...

}

Figure 1 An illustrative example of a code working with cyclic doubly-linked lists and its analysis. The C expressions like `ptr->field` can be seen as syntactic sugar for expressions using pointer arithmetic of the form `*(ptr + offsetof(field))`. The $\mathfrak{e}(X)$ predicates representing *the end of the block pointed by* X are dropped from the (P, Q) pairs for simplicity.

intuitively, means that while the addresses X and $X + 8$, which are allocated by the formulae $X \mapsto L_1$ and $X + 8 \mapsto L_2$, may – though need not – belong to a single memory block, the values stored at these addresses within the block do not overlap.[3]

The same principles are then used for the computation of the contracts for the `insert_after` and `main` functions. Here, let us just highlight a situation that happens, e.g., upon the `j->next = n` statement of `insert_after`. Notice that, in its case, the so-far computed precondition P must be extended by the new requirement $J \mapsto B_1$, stating that J must be allocated, and Q is then extended by the fact $J \mapsto N$, which is the effect of executing the given statement. At the same time, however, the rest of the previously computed symbolic state of the program Q stays untouched (in general, only some part may be preserved). Given the current symbolic state Q and a statement, the problem of deriving which precondition is missing and which part of the state will remain untouched is denoted as the *bi-abduction problem*, and a procedure looking for its solution is a *bi-abduction procedure*. The computed missing part of the pre-condition is called the *anti-frame*, and the computed part of the current symbolic state not modified by the statement being executed is called the *frame*.

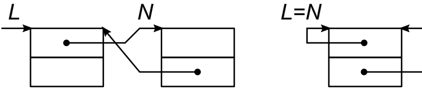
When analysing the `main` function, one does already need not re-analyse the `init_dll` and `insert_after` functions – instead, one simply uses their contracts. For simplicity, we assume here that `malloc` always succeeds, and hence even `main` is deterministic. After the execution of `malloc`, we use the special predicate $x \mapsto \top[24]$ to express that a sequence of 24 bytes of undefined contents was allocated. We allow such blocks (as well as all other kinds of blocks that arise during the analysis) be *split* to smaller parts whenever this is needed for applying a contract of some function (or statement). That happens, e.g., on lines b and d of the `main` function where the block $X \mapsto \top[24]$ created by `malloc` is split to 3 fields as described by $X \mapsto \top[8] * X + 8 \mapsto \top[8] * X + 16 \mapsto \top[8]$. The last two of the fields then match the precondition of `init_dll`, and the first one becomes a frame (untouched by the function).

Without now going into further details, we note that analysing more complex functions requires one to solve multiple more problems. For example, if there appears some non-determinism, one needs to start working with contracts with disjunctive post-conditions and even with sets of such contracts. If the code contains loops, one needs to prevent the analysis from diverging while generating more and more points-to predicates. For that, one can use widening in the form of a list abstraction. The resulting over-approximation may then, however, render some generated pre-/post-condition pairs unsound, leading to a need to run another phase of the analysis that will start from the computed pre-conditions and check, without using abduction any more, what post-condition the code can really guarantee. We discuss all these issues in the extended version of this paper [22].

However, before proceeding, let us stress how significantly the above-mentioned use of the per-field separation distinguishes our approach from its predecessor bi-abduction analysis [6, 7]. That analysis would use *whole-block* predicates of the form $X \mapsto dll(next : A, prev : B)$ to describe instances of `struct dll`, while we use the formula $X \mapsto A * X + 8 \mapsto B * \mathfrak{b}(X) = \mathfrak{b}(X + 8)$. The per-field separating conjunction allows us to (1) express partial information about a block and (2) infer a precondition where two (or more) fields can be in the same block as well as in different blocks. Point 1 helps us to generate contracts of functions where we do not know the exact sizes of the allocated block – e.g., `init_dll` does not require the

[3] In a formula $a \mapsto b * c \mapsto d$ with a per-object separating conjunction, a and c are two distinct objects allocated in memory (while b and d need not be allocated and may coincide). With a per-field separating conjunction, a and c are allowed to be non-overlapping *fields* of the same allocated object.

pointer x to point to an instance of `struct dll`, it can be, e.g., used on larger structures, such as, e.g., `struct emb_dll`, that *embed* the original structure. Point 2 is used in the contract of `insert_after` where the formula $L \mapsto N * N + 8 \mapsto B_3$ describes a memory where it may be that $L = N$ as well as $L \neq N$. The contract for `insert_after` can then be applied on a circular doubly-linked list consisting of a single item ($L = N$) as well on lists consisting of more items ($L \neq N$) – see the figure below for an illustration.



Note that when one uses the whole-block predicate, the precondition of `insert_after` in the form $L \mapsto dll(next : N, prev : _) * N \mapsto dll(next : _, prev : B_3) * J \mapsto dll(next : B_1, prev : B_2)$ requires $L \neq N$, and hence it is not covering the two above mentioned cases. One can of course sacrifice performance of the analysis and generate multiple contracts by modifying the abduction rules – e.g., one can non-deterministically introduce an alias $L = N$ before inferring the anti-frame on line v of `main` to get the pre-condition $L \mapsto dll(next : L, prev : _) * L = N$. Introducing such non-determinism is, however, costly. That is why, as we will see in our experiments, it is not done in tools such as Infer, which can then cause that such tools will miss some function contracts (or generate incomplete contracts that will not be applicable in some common cases: such as insertion into a list of length 1).

An additional example is provided in the technical report [22], where pointer arithmetic and bit-masking are directly visible in the C-code.

3 Memory Model

In the following, we introduce the memory model that we use in this paper. *Values* are sequences of bytes, i.e., $Val = Byte^+$, where bytes are 8-bit words. Sequences of bytes can be interpreted as numbers – either signed or unsigned, which we leave as a part of the operations to be applied on the sequences (including conversion operations). We designate a subset of the values $Loc = Byte^N \subseteq Val$ as *locations* where $N \geqslant 1$ is the byte-width of words of a given architecture and where byte sequences to be interpreted as locations are always understood as unsigned. The null pointer is represented by $0 \in Loc$ in our memory model.

We will use so-called *stack-block-memory triplets* (SBM triplets for short) as *configurations* of our memory model in order to define the operational semantics of programs (and also to define the semantics of our separation logic later on):

Stack. We assume some set of *variables* Var where each variable $x \in Var$ has some fixed positive size, denoted as $\mathsf{size}(x)$. Then, *Stack* is the set of total functions $Var \to Val$ such that each variable is mapped to a byte sequence whose length is according to the size of the variable, i.e., for each stack $S \in Stack$ and variable $x \in Var$, we have $S(x) \in Byte^{\mathsf{size}(x)}$.

Memory. *Mem* is the set of partial functions $Loc \rightharpoonup Byte$ that define the contents of allocated memory locations.

Blocks. We use $Interval = \{ [l, u) \mid l < u \text{ where } l, u \in Loc\}$ to denote intervals of subsequent memory locations where we include the lower bound and exclude the upper bound. Intuitively, an interval $[l, u) \in Interval$ will denote which locations were allocated at the same time (and must thus also be deallocated together, can be subtracted using pointer

subtraction, etc.). $Block = \{\, [l, u) \in \mathit{Interval} \mid l \neq 0\}$ are intervals whose lower bound is not 0 (recall that null is represented by $0 \in \mathit{Loc}$ in our memory model). $\mathit{Blocks} \subseteq (\mathbf{2}_{\mathit{fin}})^{\mathit{Block}}$ is the set of all finite sets of non-overlapping blocks, i.e., for all $B \in \mathit{Blocks}$ and for all $[l_1, u_1), [l_2, u_2) \in B$ such that either $l_1 \neq l_2$ or $u_1 \neq u_2$, we have that either $u_1 \leqslant l_2$ or $u_2 \leqslant l_1$.

Configurations. *Config* consists of all triplets $(S, B, M) \in \mathit{Stack} \times \mathit{Blocks} \times \mathit{Mem}$ such that the set of allocated blocks and the locations whose contents is defined are linked as follows:

- For every $\ell \in \mathit{Loc}$ s.t. $M(\ell)$ is defined, there is a block $[l, u) \in B$ s.t. $\ell \in [l, u)$.[4]

We introduce functions $\mathfrak{b}_B, \mathfrak{e}_B : \mathit{Loc} \to \mathit{Loc}$, parameterized by some set of blocks $B \in \mathit{Blocks}$, which return the base or end address, respectively, of the block to which a given location belongs, i.e., given some $\ell \in \mathit{Loc}$, we set $\mathfrak{b}_B(\ell) = l$ in case there is some $[l, u) \in B$ with $\ell \in [l, u)$, and $\mathfrak{b}_B(\ell) = 0$, otherwise. Likewise for $\mathfrak{e}_B(\ell)$.

Axioms. For later use, we note that, building on the above notation, we can express the requirements for locations to be within their associated block and for blocks to be non-overlapping in the form of the following two axioms:

$$\forall \ell.\ \mathfrak{b}_B(\ell) = 0 \vee \mathfrak{b}_B(\ell) \leqslant \ell < \mathfrak{e}_B(\ell)$$

$$\forall \ell, \ell'.\ (0 < \mathfrak{b}_B(\ell) < \mathfrak{e}_B(\ell') \leqslant \mathfrak{e}_B(\ell) \vee 0 < \mathfrak{b}_B(\ell') < \mathfrak{e}_B(\ell) \leqslant \mathfrak{e}_B(\ell')) \to \mathfrak{b}_B(\ell) = \mathfrak{b}_B(\ell') \wedge \mathfrak{e}_B(\ell) = \mathfrak{e}_B(\ell')$$

Notation. Given a (partial) function f, $f[a \hookrightarrow b]$ denotes the (partial) function identical to f up to $f[a \hookrightarrow b](a) = b$. Moreover, $f[a \hookrightarrow \bot]$ denotes the (partial) function identical to f up to being undefined for a.

4 A Low-level Language and Its Operational Semantics

We now state a simple low-level language together with its operational semantics. The language is close to common intermediate languages into which programs in C are compiled by compilers such as `gcc` or `clang`. We assume that a type checker ensures that variables of the right sizes are used, guaranteeing, in particular, that the left-hand side (LHS) and right-hand side (RHS) of an assignment are of the same size or that the dereference operator is only applied to locations. We do not include the operators of *item access* (`.` and `->`) nor *indexing* (`[]`) into our language as their usage can be compiled to using *pointers*, *pointer arithmetic*, and the *dereference operator* (`*`) as indeed commonly done by compilers. Likewise, we do not include the *address-of operator* (`&`) whose usage can be replaced by storing all objects whose address should be derived via `&` into dynamically allocated memory, followed by using pointers to such memory, as also done automatically by some compilers. Further, we assume the `sizeof` and `offsetof` operators be resolved and transformed to constants.

We now present the statements of our low-level language together with their operational semantics. The semantics is defined over configurations, which we introduced in the previous section. The semantics maintains the following invariant:

[4] Note that we do not require the reverse, i.e., that all locations of a block are allocated. This is because our separation logic is set up to work with partially allocated blocks. In particular, the separating conjunction needs to break up blocks into partial blocks. We note, however, that the semantics of our programming language maintains the invariant that each block is always fully allocated.

- For every $[l, u) \in B$ and every $\ell \in [l, u)$, $M(\ell)$ is defined.

We start with rules describing various *assignment statements* possibly combined with pointer dereferences either on the LHS or RHS. In the rules (and further on), we use $M[\ell, \ell')$ to denote the byte sequence $M(\ell)M(\ell+1)\cdots M(\ell'-1)$:

$$(S, B, M) \xrightarrow{x:=k} (S[x \hookrightarrow k], B, M) \text{ for some value } k \in \mathit{Val}$$

$$(S, B, M) \xrightarrow{x:=y} (S[x \hookrightarrow S(y)], B, M)$$

$$(S, B, M) \xrightarrow{x:=*y} \text{ if } \mathfrak{b}_B(S(y)) = 0 \text{ or } S(y) + \mathsf{size}(x) > \mathfrak{e}_B(S(y)), \text{ then } \mathit{error} \text{ else } (S[x \hookrightarrow M[S(y), S(y) + \mathsf{size}(x))], B, M)$$

Note that, in the case of $x := *y$, one needs to read $\mathsf{size}(x)$ bytes from the adress $S(y)$. This is impossible if the condition $S(y) + \mathsf{size}(x) > \mathfrak{e}_B(S(y))$ holds.

$$(S, B, M) \xrightarrow{*x:=y} \text{ if } \mathfrak{b}_B(S(x)) = 0 \text{ or } S(x) + \mathsf{size}(y) > \mathfrak{e}_B(S(x)), \text{ then } \mathit{error} \text{ else } (S, B, M[[S(x), S(x) + \mathsf{size}(y)) \hookrightarrow S(y)])$$

We continue by *memory allocation*. We treat 0-sized allocations as an error.[5] For non-zero-sized allocations, the allocation can always fail and return null, otherwise the successfully allocated memory block is initialized with some arbitrary value[6]:

$$(S, B, M) \xrightarrow{x=malloc(z)} \text{ if } S(z) = 0 \text{ then } \mathit{error} \text{ else either } (S[x \hookrightarrow \mathsf{null}], B, M) \text{ or}$$
$$(S[x \hookrightarrow \ell], B \cup \{[\ell, \ell + S(z))\}, M[[\ell, \ell + S(z)) \hookrightarrow k]) \text{ for some } k \in \mathit{Byte}^{S(z)} \text{ and } \ell > 0$$
$$\text{such that } \ell + S(z) \leqslant 2^{8N} \text{ and } [\ell, \ell + S(z)) \text{ does not overlap with any } [l, u) \in B$$

The calloc function, which nullifies the allocated block, can be defined analogically to `malloc`, by just changing $M[[\ell, \ell + S(z)) \hookrightarrow k]$ to $M[[\ell, \ell + S(z)) \hookrightarrow 0^{S(z)}]$. The `realloc` function, which shrinks or enlarges a block, possibly moving it to a different memory location, can be reduced to a sequence of other statements, and so we do not introduce it explicitly for brevity.

The *deallocation* of memory is modelled by the following rule:[7]

$$(S, B, M) \xrightarrow{free(x)} \text{ if } S(x) \neq \mathfrak{b}_B(S(x)) \text{ then } \mathit{error} \text{ else } (S, B \backslash \{[S(x), \mathfrak{e}_B(S(x)))\}, M[[S(x), \mathfrak{e}_B(S(x))) \hookrightarrow \bot])$$

The low-level language further contains a collection of binary and unary operations denoted as bop and uop, respectively. The operations of adding an offset to a pointer (ptrplus) and pointer subtraction (ptrsub) are special and handled separately. The operation ptrplus for *adding a (possibly negative) offset to a pointer* requires its pointer argument to be defined,

[5] The C standard says that the behaviour in this case is user-defined, the allocation can return null or a non-null value, which, however, cannot be dereferenced. However, since such an allocation is usually suspicious, many analysers flag it as an error/warning. We adopt the same approach, but if need be, the rules could be changed to handle such allocations according to the standard.

[6] Notice that 2^{8N} gives the largest address that can be expressed using words with the byte-width N.

[7] Notice that we do not need a rule for deallocating zero-sized blocks since we do not allow such blocks to be created.

and, in accordance with the C standard, the result must be within the appropriate memory block plus one byte (i.e., it may point just behind the end of the block).[8] The operation ptrsub for *pointer subtraction* is special in that it requires its pointer operands to be defined, to have the same base, and to point inside an allocated block or just behind its end. We also support the `memcpy` statement (and can simulate the `memmove` statement). To encode *conditional branching* arising from conditional statements or loops, we introduce the `assume` statement that models conditions $x \bowtie y$ for $\bowtie \in \{=, \neq, \leqslant, <, \geqslant, >\}$. We allow *functions* without a return value, not referring to global variables, having parameters passed by reference only, with the names of the parameters unique to each function, and not having local variables. We also introduce the `assert` statement that is similar to the `assume` statement, but it checks at runtime whether the specified condition holds, and it fails if this is not the case. The operational semantics of all these statements can be found in [22].

5 Separation Logic

We now introduce a separation logic that supports reasoning about low-level memory models as introduced earlier. Our separation logic (SL) has the following syntax:

$$\begin{aligned}\varphi \ ::= \ & \varepsilon_1 \mapsto \varepsilon_2 \mid \varepsilon_1 \mapsto k[\varepsilon_2] \mid \varepsilon_1 \mapsto \top[\varepsilon_2] \mid \varphi_1 * \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2) \mid \\ & \mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2') \mid \mathsf{emp} \mid \mathit{true} \mid \varepsilon_1 \bowtie \varepsilon_2 \mid \exists x.\varphi \\ \bowtie \ ::= \ & = \mid \neq \mid \leqslant \mid < \mid \geqslant \mid > \qquad \varepsilon \ ::= \ k \mid x \mid \mathfrak{b}(\varepsilon) \mid \mathfrak{e}(\varepsilon) \mid \mathsf{uop}\,\varepsilon \mid \varepsilon_1\,\mathsf{bop}\,\varepsilon_2\end{aligned}$$

Variables and Values. Our SL formulae are stated over the same set of variables *Var* and values *Val* that we introduced in the definition of our memory model. In particular, the variables x, y, z and the values k of our SL formulae are drawn from *Var* and *Val*, respectively.

Size. Variables, values, operators, and expressions in our logic are typed by their *size*. We will only work with formulae where the variables and values respect the sizes expected by the involved operations and predicates. For every expression ε, we denote by $\mathsf{size}(\varepsilon)$ the size of the value to which this expression may evaluate. We remark on the choice of working with fixed sizes: We intentionally do not permit variables of variable size because (1) such variables are typically not supported by low-level languages and (2) variables of variable size allow one to model strings, which would make our language vastly more powerful (allowing one to model all kinds of string operations)[9].

Points-To Predicates. The points-to predicate $\varepsilon_1 \mapsto \varepsilon_2$ denotes that the byte sequence ε_2 is stored at the memory location ε_1. Due to we are working with expressions of fixed size, every model of $\varepsilon_1 \mapsto \varepsilon_2$ must allocate exactly $\mathsf{size}(\varepsilon_2)$ bytes. In addition, we introduce two restricted cases of points-to predicates where the RHS is of parametric size: namely, $\varepsilon_1 \mapsto k[\varepsilon_2]$ and $\varepsilon_1 \mapsto \top[\varepsilon_2]$ that allow us to say that ε_1 points to an array of ε_2 bytes that either all have the same constant value k or have any value, respectively. These predicates allow us to, e.g., express that some block of memory is nullified, which is often crucial to

[8] We are aware that this requirement is not respected in some real-life pograms, such as, e.g., the implementation of lists in Linux. We will later mention that our approach can be relaxed to handle such cases too.

[9] We believe that extending our later presented analysis to such variables is possible (by recording the length of the target object as another parameter of the points-to predicate), but we leave it for future work in order not to complicate the basic approach we propose.

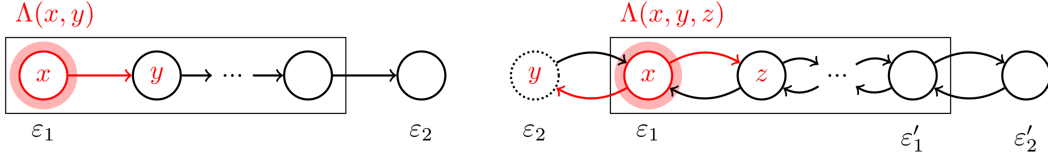


Figure 2 An illustration of the meaning of the $\mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2)$ and $\mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$ formulae.

know when analysing advanced implementations of dynamic data structures [16]. We lift the notion of size to the RHS of these points-to predicates as follows: $\mathsf{size}(k[y]) = \mathsf{size}(\top[y]) = y$. In $\varepsilon_1 \mapsto k[\varepsilon_2]$, we require k to be a single byte, i.e., $\mathsf{size}(k) = 1$.

Notation. Given a formula φ, we write $var(\varphi)$ to denote the *free* variables of φ (as usual a variable is *free* if it does appear within an existential quantification). Further, given an expression ε, we write $var(\varepsilon)$ for all variables appearing in ε.

Terminology. We call formulae that do not contain the disjunction operator ($\vee$) *symbolic heaps*. We will mostly work with symbolic heaps in this paper. Disjunctions of symbolic heaps will be only used on the RHS of (some) contracts. We call formulae that do not contain existential quantification ($\exists$) *quantifier-free*. Our SL contains the relational predicates $\varepsilon_1 \bowtie \varepsilon_2$, which include equality and disequality; these predicates are traditionally called *pure* in the separation logic literature. We follow this terminology and call any separating conjunction of such predicates a *pure formula*.

List-Segment Predicates. List segments in our SL are parameterized by a *segment* predicate $\Lambda(x, y)$ or $\Lambda(x, y, z)$ for singly-linked or doubly-linked lists, respectively; see Fig. 2 for an illustration of the semantics of $\mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2)$ and $\mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$ for $\Lambda(x, y) \equiv x \mapsto y$ and $\mathsf{dls}_{\Lambda(x,y,z)} \equiv x \mapsto z * x + 8 \mapsto y$. We note that our list-segment predicates only have two or three free variables, respectively, which prevents the logic from, e.g., describing non-global heap objects shared by list elements. However, more parameters could be introduced in a similar fashion to other works [2]. We have not done so here since it would complicate the notation, and we take this issue as orthogonal to the techniques we propose.

Binary and Unary Operators. uop and bop denote some arbitrary set of binary and unary operators, respectively. We assume this set to include at least the usual operators ($+$, $-$, $*$, &, |, ...) available in low-level languages as well as a special substring operator $\cdot[\cdot,\cdot)$ on byte sequences where $k[i, j)$ for some $k = b_0 \cdots b_{l-1} \in Byte^l$ and $0 \leqslant i \leqslant j \leqslant l$ denotes the byte sequence $b_i \cdots b_{j-1}$. Since we work with variables of fixed size, we basically assume a version of each uop and bop for every possible operand size. We further remark that unary operators uop can be used for modelling the casting to different sizes.

Semantics. We now define the semantics of our SL over SBM triplets $(S, B, M) \in Config$:

$(S, B, M) \models \varepsilon_1 \mapsto \varepsilon_2$ iff

$$\mathsf{dom}(M) = [[\![\varepsilon_1]\!]_{S,B}, [\![\varepsilon_1]\!]_{S,B} + \mathsf{size}(\varepsilon_2)) \text{ and } M[[\![\varepsilon_1]\!]_{S,B}, [\![\varepsilon_1]\!]_{S,B} + \mathsf{size}(\varepsilon_2)) = [\![\varepsilon_2]\!]_{S,B}$$

where

$$[\![k]\!]_{S,B} = k, [\![x]\!]_{S,B} = S(x), [\![\mathfrak{b}(\varepsilon)]\!]_{S,B} = \mathfrak{b}_B([\![\varepsilon]\!]_{S,B}), [\![\mathfrak{e}(\varepsilon)]\!]_{S,B} = \mathfrak{e}_B([\![\varepsilon]\!]_{S,B}),$$
$$[\![\mathsf{uop}\,\varepsilon]\!]_{S,B} = \mathsf{uop}([\![\varepsilon]\!]_{S,B}), \text{ and } [\![\varepsilon_1 \,\mathsf{bop}\, \varepsilon_2]\!]_{S,B} = [\![\varepsilon_1]\!]_{S,B} \;\mathsf{bop}\; [\![\varepsilon_2]\!]_{S,B}$$

$(S,B,M) \models \varepsilon_1 \mapsto k[\varepsilon_2]$ iff
$\mathsf{dom}(M) = [[\![\varepsilon_1]\!]_{S,B}, [\![\varepsilon_1]\!]_{S,B} + [\![\varepsilon_2]\!]_{S,B})$ and $M[[\![\varepsilon_1]\!]_{S,B} + i] = k$ for all $0 \leqslant i < [\![\varepsilon_2]\!]_{S,B}$

$(S,B,M) \models \varepsilon_1 \mapsto \top[\varepsilon_2]$ iff $\mathsf{dom}(M) = [[\![\varepsilon_1]\!]_{S,B}, [\![\varepsilon_1]\!]_{S,B} + [\![\varepsilon_2]\!]_{S,B})$

We remark on the difference between the three points-to predicates: the predicate $\varepsilon_1 \mapsto \varepsilon_2$ fixes the exact sequence of bytes ε_2 that is stored from location ε_1 onwards, and the number of bytes is known (the size of ε_2); the predicate $\varepsilon_1 \mapsto k[\varepsilon_2]$ states that there are ε_2 number of bytes stored from location ε_1 onwards (note that the number of bytes ε_2 is symbolic), and each of these bytes equals k; and the predicate $\varepsilon_1 \mapsto \top[\varepsilon_2]$ works in the same way except that the bytes stored are not fixed.

$(S,B,M) \models \varphi_1 * \varphi_2$ iff there are some M_1, M_2 with $M = M_1 \uplus M_2, (S,B,M_i) \models \varphi_i$

$(S,B,M) \models \varphi_1 \vee \varphi_2$ iff $(S,B,M) \models \varphi_1$ or $(S,B,M) \models \varphi_2$

$(S,B,M) \models \mathsf{emp}$ iff $\mathsf{dom}(M) = \varnothing$ $\qquad$ $(S,B,M) \models \mathit{true}$ always holds

$(S,B,M) \models \varepsilon_1 \bowtie \varepsilon_2$ iff $\mathsf{dom}(M) = \varnothing$ and $[\![\varepsilon_1]\!]_{S,B} \bowtie [\![\varepsilon_2]\!]_{S,B}$

We point out that pure formulae constrain the heap to be empty. This is typically not required by separation logics that support classical (non-separating) conjunction at least on pure sub-formulae. However, we exclude the classical conjunction in order to simplify the presentation and hence need to constrain the heap of pure formulae to be empty.

$(S,B,M) \models \exists x.\varphi(x)$ iff there is some $v \in \mathit{Val}$
and a fresh variable $u \in \mathit{Var}$ s.t. $(S[u \hookrightarrow v], B, M) \models \varphi(u)$

$(S,B,M) \models \mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2)$ iff $(S,B,M) \models \varepsilon_1 = \varepsilon_2$ or
$(S,B,M) \models \varepsilon_1 \neq \varepsilon_2 * \mathsf{true}$ and there is some $\ell \in \mathit{Loc}$
and a fresh variable $u \in \mathit{Var}$ s.t. $(S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u) * \mathsf{ls}_{\Lambda(x,y)}(u, \varepsilon_2)$

$(S,B,M) \models \mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$ iff $(S,B,M) \models \varepsilon_1 = \varepsilon_2' * \varepsilon_2 = \varepsilon_1'$ or
$(S,B,M) \models \varepsilon_1 \neq \varepsilon_2' * \varepsilon_2 \neq \varepsilon_1' * \mathsf{true}$ and there is some $\ell \in \mathit{Loc}$ and a fresh variable
$u \in \mathit{Var}$ such that $(S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u, \varepsilon_2) * \mathsf{dls}_{\Lambda(x,y,z)}(u, \varepsilon_1, \varepsilon_1', \varepsilon_2')$

Satisfiability and Entailment. We say that an SL formula φ is *satisfiable* iff there is a model (S,B,M) such that $(S,B,M) \models \varphi$. We say that an SL formula φ_1 *entails* an SL formula φ_2, denoted $\varphi_1 \models \varphi_2$, iff we have that $(S,B,M) \models \varphi_2$ for every model (S,B,M) such that $(S,B,M) \models \varphi_1$.

Restrictions on the Segment Predicates. From now on, we put further restrictions on the segment predicates $\Lambda(x, y)$ and $\Lambda(x, y, z)$: (1) Λ needs to be of the shape $\exists x_1, \ldots, x_k.\varphi$ for some quantifier-free symbolic heap φ. Intuitively, this condition is required since quantifier-free symbolic heaps are the formulae on which the symbolic execution described in Section 6 is based on and the existential quantification allows to hide some nested data. (2) Λ needs to be *block-closed* in the sense defined below.

Block-closedness. A formula φ is *block-closed* iff, for all $(S, B, M) \models \varphi$ and $\ell \in \mathsf{dom}(M)$, we have that $[\mathfrak{b}(\ell), \mathfrak{e}(\ell)) \subseteq \mathsf{dom}(M)$. Intuitively, block-closedness ensures that all points-to assertions in a formula add up to whole blocks. We require block-closedness in order to ensure that list-segments correspond to our intuition and connect different memory blocks (i.e., we exclude models where multiple or all nodes of list-segments belong to the same block). Technically, the requirement of block-closedness makes it easier to formulate rules for materialisation of list-segment nodes in the abduction procedure and for entailment checking. We leave lifting the restriction of block-closedness for future work. A sufficient condition for block-closedness, which is easy to check, is that all points-to assertions in φ can be organized in groups $\varepsilon_i \mapsto \Upsilon_i$, for $1 \leqslant i \leqslant n$, where Υ represents either y, $k[y]$, or $\top[y]$, such that $\varepsilon_i = \varepsilon_{i-1} + \mathsf{size}(\Upsilon_i)$ for all $1 < i \leqslant n$, and φ implies that $\mathfrak{e}(\varepsilon_1) - \mathfrak{b}(\varepsilon_1) = \sum_{i=1..n} \mathsf{size}(\Upsilon_i)$.

6 Contracts of Functions and Their Generation

Our analysis is based on generating *contracts of functions* along the call tree, starting from its leaves. The contracts summarize the semantics of the functions under analysis. We may also compute multiple contracts for the same function where each contract provides a valid summary of the function; the contracts might, however, differ in the preconditions under which they apply.

6.1 Contracts of Functions

We assume a set of *variables* $Var = PVar \uplus LVar$ that is partitioned into two disjoint infinite set of *program variables* $PVar$ and *logical variables* $LVar$ (also called *ghost* variables). For functions $f(x_1, \ldots, x_n)$ with parameters x_i, we always require $x_1, \ldots, x_n \in PVar$ (we assume that $x_1, \ldots, x_n$ are the only variables occurring in the body of f). To summarize the semantics of a function $f(x_1, \ldots, x_n)$, we use (sets of) *contracts* of the form $\{P\} f(x_1, \ldots, x_n)\{Q\}$ where

- the *pre-condition* P is a quantifier-free symbolic heap, and
- the *post-condition* Q is a disjunction of formulas of the form $\exists U_Q.(Q_{free} * Q_{eq})$ such that Q_{free} is a quantifier-free symbolic heap with $var(Q_{free}) \subseteq LVar$, Q_{eq} is the formula $x_1 = \varepsilon_1 * \cdots * x_n = \varepsilon_n$ for some expressions ε_i with $var(\varepsilon_i) \subseteq LVar$, and $U_Q = (var(Q_{free} * Q_{eq}) \cap LVar) \setminus var(P)$. Note that every disjunct of the post-condition Q describes the heap by a formula over the logical variables (the formula Q_{free}) and fixes the values of the program variables in terms of expressions over the logical variables (the formula Q_{eq}) where all logical variables that do not appear in the pre-condition P are existentially quantified (on the other hand, those logical variables that appear in the pre-condition may be implicitly considered as universally quantified).
- We call a contract *conjunctive* if the post-condition $Q \equiv Q_1 \vee \cdots \vee Q_l$ consists of a single disjunct (i.e., $l = 1$), and *disjunctive* otherwise.

Soundness of contracts. We will now state what it means for a contract to be sound. As usual we stipulate that configurations satisfying the pre-condition lead to configurations satisfying the post-condition. In addition, we also require that we can always add a *frame*

to the pre-/post-condition, i.e., a formula describing a part of the heap untouched by the function[10]. Here, a frame F is any symbolic heap with $var(F) \subseteq LVar$. A contract $\{P\}f(x_1,\ldots,x_n)\{Q\}$ is called *sound* iff, for all frames F, all triples (S,B,M) such that $(S,B,M) \models F * P$, and all executions of $f(x_1,\ldots,x_n)$ that start from (S,B,M) and end in some configuration (S',B',M')[11], it holds that $(S',B',M') \models F * Q$.

6.2 Contracts for Basic Statements

We give below contracts for the basic statements of our programming language stated as functions (basic statements may be viewed as special built-in functions). For simplicity (and w.l.o.g.), we assume that it never happens that the same variable appears both at the LHS and RHS of an assignment[12]. Recall that emp is implicit in all otherwise pure constraints (and so we do not need to repeat it):

- Function $\mathtt{assign}(x,y)$ with the body $x := y$:

  $$\{y = Y\}\ \mathtt{assign}(x,y)\ \{x = Y\ *\ y = Y\}.$$

- Function $\mathtt{const}_k(x)$ with the body $x := k$:

  $$\{\mathsf{emp}\}\ \mathtt{const}_k(x)\ \{x = k\}.$$

- Function $\mathtt{load}(x,y)$ with the body $x := *y$:

  $$\{y = Y\ *\ Y \mapsto z\}\ \mathtt{load}(x,y)\ \{x = z\ *\ y = Y\ *\ Y \mapsto z\}$$

  with $Q_{free} \equiv Y \mapsto z$ and $Q_{eq} \equiv x = z\ *\ y = Y$.
- Function $\mathtt{store}(x,y)$ with the body $*x := y$:

  $$\{x = X\ *\ y = Y\ *\ X \mapsto z\}\ \mathtt{store}(x,y)\ \{x = X\ *\ y = Y\ *\ X \mapsto Y\}$$

  with $Q_{free} \equiv X \mapsto Y$ and $Q_{eq} \equiv x = X\ *\ y = Y$.
- Function $\mathtt{malloc}(x,y)$ that either succeeds or fails to allocate memory through $x := \mathtt{malloc}(y)$:

  $$\{y = Y\}\ \mathtt{malloc}(x,y)\ \{x = \mathsf{null}\ *\ y = Y \vee \exists u.\ x = u\ *\ \nu(u,Y)\ *\ y = Y\}$$

  where $\nu(u,Y) = u \mapsto \top[Y] * \mathfrak{b}(u) = u\ *\ \mathfrak{e}(u) = u + Y$. Note that either $Q_{free} \equiv \nu(u,Y)$ and $Q_{eq} \equiv x = u\ *\ y = Y$ or $Q_{free} \equiv \mathsf{emp}$ and $Q_{eq} \equiv x = \mathsf{null}\ *\ y = Y$. A very similar contract can be used for calloc, just with $u \mapsto \top[Y]$ changed to $u \mapsto 0[Y]$. We remark that the contracts for $\mathtt{malloc}$ and calloc are the only disjunctive contracts among the contracts for the basic statements of our programming language.
- Function $\mathtt{free}(x)$ called with the null argument:

  $$\{x = X\ *\ X = \mathsf{null}\}\ \mathtt{free}(x)\ \{x = X\ *\ X = \mathsf{null}\}$$

- Function $\mathtt{free}(x)$ called over a non-null argument:

  $$\{x = X\ *\ X \mapsto \top[y] * \mathfrak{b}(X) = X\ *\ \mathfrak{e}(X) = X + y\}\ \mathtt{free}(x)\ \{x = X\}$$

[10] That is, we directly incorporate the well-known *frame rule* from the separation-logic literature into our notion of soundness. We choose to do so for economy of exposition and for making the paper self-contained. As an alternative one could derive the validity of the frame rule from the fact that all contracts of the basic statements, as stated in Section 6.2, are *local actions* in the sense of [8] (which is equivalent to Lemma 1 stated in this paper).

[11] Note that $\mathsf{dom}(S') = \mathsf{dom}(S)$ and that we have $S'(x) = S(x)$ for all $x \in LVar$ because logical variables do not occur in the program and hence are never updated.

[12] We may assume this because assignments such as $x := *x$ can always be rewritten to the sequence $y := *x; x := y$ (at the cost of introducing a fresh variable y).

Note that a block to be freed may be split into multiple fields at the time of freeing. We, however, do not need to deal with this issue here since the later presented bi-abduction rules will split the LHS of the contract of `free` such that it can match the fragmented block.

- Functions $\texttt{assign}_{\mathsf{bop}}(x, y, z)$ with the body $x := y \,\mathsf{bop}\, z$ for binary operators bop (and likewise for unary operators uop):

$$\{y = Y \ * \ z = Z\}\ x := y \,\mathsf{bop}\, z\ \{x = Y \,\mathsf{bop}\, Z \ * \ y = Y \ * \ z = Z\}$$

- Function $\mathsf{ptrplus}(x, y, z)$ with the body $x := y \,\mathsf{ptrplus}\, z$ for the case when the result is within the block of the pointer to which an offset is added:

$$\{y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}\ x := y \,\mathsf{ptrplus}\, z\ \{x = Y + Z \ * \ y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}$$

for $\varphi_{Y,Z} \equiv \mathfrak{b}(Y) \neq 0 \ * \ \mathfrak{b}(Y) = \mathfrak{b}(Y + Z) * \mathfrak{e}(Y) = \mathfrak{e}(Y + Z)$.

- Function $\mathsf{ptrplus}(x, y, z)$ with the body $x := y \,\mathsf{ptrplus}\, z$ for the case when the result points one byte past the block of the pointer to which an offset is added:

$$\{y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}\ x := y \,\mathsf{ptrplus}\, z\ \{x = Y + Z \ * \ y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}$$

for $\varphi_{Y,Z} \equiv \mathfrak{b}(Y) \neq 0 \ * \ Y + Z = \mathfrak{e}(Y)$.

- Function $\mathsf{ptrsub}(x, y, z)$ with the body $x := y \,\mathsf{ptrsub}\, z$:

$$\{y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}\ x := y \,\mathsf{ptrsub}\, z\ \{x = Y - Z \ * \ y = Y \ * \ z = Z\}$$

for $\varphi_{Y,Z} \equiv \mathfrak{b}(Y) \neq 0 \ * \ \mathfrak{b}(Y) \leqslant Z \leqslant \mathfrak{e}(Y)$.

- Function $\texttt{assume}_{\bowtie}(y, z)$ with the body $\texttt{assume}(y \bowtie z)$:

$$\{y = Y \ * \ z = Z\}\ \texttt{assume}(y \bowtie z)\ \{y = Y \ * \ z = Z \ * \ y \bowtie z\}$$

- Function $\texttt{assert}_{\bowtie}(y, z)$ with the body $\texttt{assert}(y \bowtie z)$:

$$\{y = Y \ * \ z = Z \ * \ y \bowtie z\}\ \texttt{assert}(y \bowtie z)\ \{y = Y \ * \ z = Z \ * \ y \bowtie z\}$$

Finally, the contract for `memcpy` is more complex, and we defer it to [22] for space reasons. We now state the soundness of the contracts for the basic statements of our programming language:

▶ **Lemma 1.** *Let stmt be a basic statement and let $\{P\}\ f(x_1, \ldots, x_n)\ \{Q\}$ be a contract for stmt as stated above. Then, the contract is sound, i.e., for all frames F, all configurations (S, B, M) such that $(S, B, M) \models F * P$, and all executions of $f(x_1, \ldots, x_n)$ that start from (S, B, M) and end in some configuration (S', B', M'), it holds that $(S', B', M') \models F * Q$.*

Proof. Direct from the semantics of our programming language as stated in Section 4. ◀

6.3 Contract Generation

We now sketch the generation of contracts for an arbitrary user-defined function $f(x_1, \ldots, x_n)$. Our analysis proceeds along the call tree, starting from its leaves. Hence, we can assume to already have computed contracts for nested function calls. (Recall that, in this paper, we limit ourselves to non-recursive functions.) We derive contracts by (forward) symbolic execution. The symbolic execution starts at the beginning of f and maintains a pair of formulae P and Q, representing the so-far computed part of the *pre-condition* of the function f and the *current symbolic state*. The symbolic execution will guarantee that configurations that satisfy P lead to configurations satisfying Q after executing the so-far analysed statements. P and Q will change throughout the symbolic execution because we keep restricting the

precondition P and advancing the symbolic state Q. The symbolic execution is set up such that the program variables $x_1, \ldots, x_n$ may be updated, while all other variables will never be modified (but, of course, fresh variables may be introduced and assigned at any time). The symbolic execution is initialised by introducing fresh logical variables $X_1, \ldots, X_n$ and setting $P \equiv Q \equiv x_1 = X_1 * \cdots * x_n = X_n$. In each step, the symbolic execution needs to solve a bi-abduction problem in order to advance the symbolic state Q with regard to the contract of a function call or a basic statement. The bi-abduction procedure might discover that the current symbolic state Q does not suffice to safely call the function, in which case either a strengthening of the precondition P is returned or the procedure fails. We describe our procedure for solving the abduction problem (the procedure for discovering missing pre-conditions) in the next section, and refer the reader to our technical report [22] for the full bi-abduction procedure. In order to derive sound contracts, we follow the two-round analysis approach of [7]: The first round (called `PreGen` in [7]) infers a set of pre-/post-condition pairs (P, Q), but there is no guarantee about the soundness of the inferred (P, Q). For each pre-/post-condition pair (P, Q) computed in the first round, the second round (called `PostGen` in [7]) discards the post-condition Q and re-starts the symbolic execution from the pre-condition P *not allowing the strengthening of the pre-condition throughout the symbolic execution*, which either fails or results in a set of pre-/post-condition pairs $(P, Q_1), \ldots, (P, Q_l)$. In the latter case, we return $(P, Q_1 \vee \cdots \vee Q_l)$, which is guaranteed to be a sound contract.

We refer an interested reader to our technical report [22] for the details on how we implement the two-round analysis of [7] and for accompanying examples.

7 Bi-Abduction Procedure

We now state our rules for computing a solution to the abduction problem. In the below rules, we will use the notation $\varphi * [M] \rhd \psi$ to denote that we are deriving the solution M to the abduction problem $\varphi * [?] \models \psi$. The rules are to be applied in the stated order.[13]

We start with a rule allowing us to learn *missing pure constraints*.

$$\texttt{learn-pure}\ \frac{\varphi * \pi * [M] \ \rhd\ \psi}{\varphi * [\pi * M] \ \rhd\ \psi * \pi}\ \pi \text{ pure}$$

The `match` rule presented below allows one to *match points-to predicates* from the LHS and RHS that have the same source location ($\varepsilon = \varepsilon'$) and points-to fields ζ, ζ' of the same size. Then we learn that the target fields are the same too. We note here that this rule is as a special case of the `split-pt-pt-right` rule presented further on, but we show it here as an easy case to start from. We discharge entailment checks of the form $\varphi_1 \models \varphi_2 * true$ where φ_2 is a pure formula (e.g. $\varepsilon = \varepsilon'$) by checking unsatisfiability of the formula $\psi_1 \wedge \neg\psi_2$ where ψ_i is a translation of the SL formula φ_i to bitvector logic. The translation procedure is sketched in our technical report [22].

$$\texttt{match}\ \frac{\varphi * \zeta = \zeta' * [M] \ \rhd\ \psi}{\varphi * \varepsilon \mapsto \zeta * [\zeta = \zeta' * M] \ \rhd\ \psi * \varepsilon' \mapsto \zeta'}\ \mathsf{size}(\zeta) = \mathsf{size}(\zeta') \text{ and } \varphi \models \varepsilon = \varepsilon' * true$$

[13] As for non-determinism within single rules, which can sometimes be applied in multiple ways, our implementation currently uses the first applicable option (with backtracking to the other options only in case that the first option turns out to result in an unsatisfiable abduction strategy). A better strategy is an open question for future research.

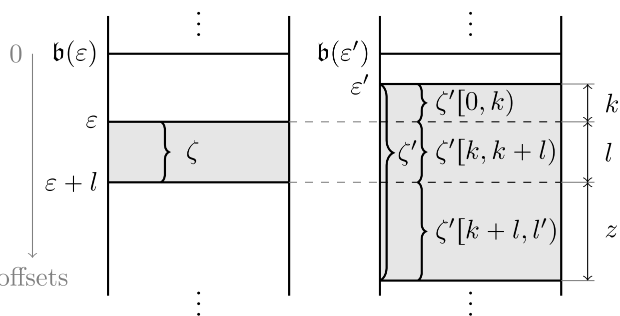


Figure 3 An illustration of the `split-pt-pt-right` rule where $z = l' - k - l$.

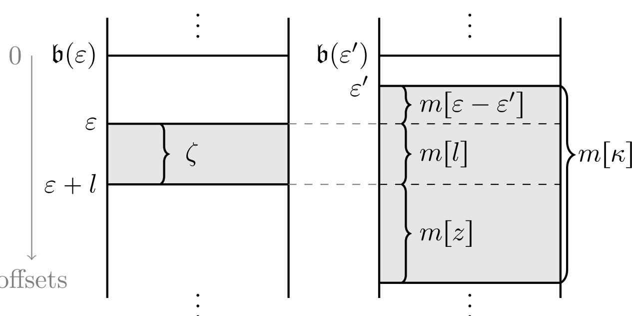


Figure 4 An illustration of the `split-pt-bl-right` rule where $z = \kappa - (\varepsilon - \varepsilon') - l$.

As illustrated in Fig. 3, the next presented `split-pt-pt-right` rule allows one to deal with pointers ε, ε' to fields ζ, ζ' that lie at possibly different addresses but within blocks of the same base address. Moreover, the RHS target field ζ' can be larger. In this case, the field ζ' is *split* to three *byte sequences* $\zeta'[0,k)$, $\zeta'[k,k+l)$, and $\zeta'[k+l,l')$, some of which can be empty, and the middle byte sequence is matched with the LHS target field ζ. (We recall that $k[i,j)$ denotes the substring of k that starts at index i and ends at index j.)

$$\texttt{split-pt-pt-right}\ \frac{\varphi * \zeta = \zeta'[k,k+l) * [M] \ \rhd\ \psi * \varepsilon' \mapsto \zeta'[0,k) * (\varepsilon + l) \mapsto \zeta'[k+l,l')}{\varphi * \varepsilon \mapsto \zeta * [\zeta = \zeta'[k,k+l) * M] \ \rhd\ \psi * \varepsilon' \mapsto \zeta'}\ C$$

In the above rule, the condition C requires that there are some $k,l,l' \in \mathbb{N}$ with $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon = \varepsilon' + k \ * \ true$, $\mathsf{size}(\zeta) = l$, $\mathsf{size}(\zeta') = l'$, and $l + k \leqslant l'$. We note that, in the above formulation of the rule `split-pt-pt-left`, we assume $0 < k$ and $k + l < l'$ in order to avoid cluttering the rule by additional case distinctions; in the case of $0 = k$ or $k + l = l'$, however, we need to remove $\varepsilon' \mapsto \zeta'[0,k)$ or $(\varepsilon + l) \mapsto \zeta'[k+l,l')$, respectively, from the RHS of the premise of the rule. There is a symmetric rule `split-pt-pt-left` for the LHS.

The `split-pt-bl-right` rule presented below and illustrated in Fig. 4 is an analogy of the rule `split-pt-pt-right` presented above, but, this time, with the RHS field, which is being split, of non-constant size. The rule covers both types of such fields that we allow: sequences of bytes of undefined values (then $m = \top$ in the rule) or sequences of the same byte (then $m \in Byte$).

$$\texttt{split-pt-bl-right}\ \frac{\varphi * \chi * [M] \ \rhd\ \psi * \varepsilon' \mapsto m[\varepsilon - \varepsilon'] * (\varepsilon + l) \mapsto m[z] * K}{\varphi * \varepsilon \mapsto \zeta * [\chi * M] \ \rhd\ \psi * \varepsilon' \mapsto m[\kappa]}\ C$$

Above, we require that $m = \top$ and $\chi \equiv \mathsf{emp}$, or $m \in \mathit{Byte}$ and $\chi \equiv \zeta = m^l$. Further, $\mathsf{size}(\zeta) = l$, C requires that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon' \leqslant \varepsilon \ * \ \varepsilon + l \leqslant \varepsilon' + \kappa \ * \ \mathit{true}$, z is some fresh variable with $\mathsf{size}(z) = N$, and $K \equiv z = \kappa - (\varepsilon - \varepsilon') - l$. There is a symmetric rule `split-pt-bl-left` for the LHS.

We now present an analogy of the above rule for the case when we need to split a field of constant size that appears on the RHS. In order to be able to split the RHS field we will also require the LHS field to be of constant size.

$$\texttt{split-bl-pt-right}\ \frac{\varphi * \chi * [M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'[0,k) * (\varepsilon + l) \mapsto \zeta'[k+l, l')}{\varphi * \varepsilon \mapsto m[\kappa] * [\chi * M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'}\ C$$

In the above rule, the condition C requires that there are some $k, l, l' \in \mathbb{N}$ with $\varphi \models \kappa = l \ * \ \mathit{true}$, $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon = \varepsilon' + k \ * \ \mathit{true}$, $\mathsf{size}(\zeta') = l'$, and $k + l \leqslant l'$. In the rule, either $m = \top$ and $\chi \equiv \mathsf{emp}$, or $m \in \mathit{Byte}$ and $\chi \equiv \zeta'[k, k+l) = m^l$. There is a symmetric rule `split-bl-pt-left` for the LHS.

We are finally getting to the `split-bl-bl-right` rule that matches two fields that are both of non-constant sizes while splitting the RHS field if need be.

$$\texttt{split-bl-bl-right}\ \frac{\varphi * [M] \ \rhd \ \psi * \varepsilon' \mapsto m'[\varepsilon - \varepsilon'] * \varepsilon + \kappa \mapsto m'[z] * K}{\varphi * \varepsilon \mapsto m[\kappa] * [M] \ \rhd \ \psi * \varepsilon' \mapsto m'[\kappa']}\ C$$

In the rule, either $m' = \top$ or $m = m'$. Further, C is the condition that requires $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon' \leqslant \varepsilon \ * \ \varepsilon + \kappa \leqslant \varepsilon' + \kappa' \ * \ \mathit{true}$ and $K \equiv z = \kappa' - (\varepsilon - \varepsilon') - \kappa$. As before, there is also a symmetric rule `split-bl-bl-left` for splitting on the LHS.

Next, we present a rule that allows one to match a points-to predicate on the LHS against a singly-linked list segment on the RHS. In fact, the rule does not directly perform the matching, but it facilitates it by *materialising* the first cell out of the list segment. The matching itself (possibly combined with splitting) is then performed by the above rules. We expect that the cells of the list segment are described using a formula of the form $\Lambda(x,y) \equiv \exists u_1, \ldots, u_k . \lambda(x, y, u_1, \ldots, u_k)$.

$$\texttt{slseg-pt-ls-right}\ \frac{\varphi * \varepsilon \mapsto \zeta * [M] \ \rhd \ \psi * \lambda[\varepsilon'/x, z/y, z_1/u_1, \ldots, z_k/u_k] * \mathsf{ls}_{\Lambda(x,y)}(z, \zeta')}{\varphi * \varepsilon \mapsto \zeta * [M] \ \rhd \ \psi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon', \zeta')}\ C$$

Above, C is the condition that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \mathit{true}$ and $z, u_1, \ldots, u_k$ are some fresh variables.

We next present a version of the above rule for the case of a list segment on the LHS. Note that, in this case, we must require the list segment be non-empty. In the rule, C is the condition that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon \neq \zeta \ * \ \mathit{true}$ and $z, u_1, \ldots, u_k$ are some fresh variables.

$$\texttt{slseg-pt-ls-left}\ \frac{\varphi * \lambda[\varepsilon/x, z/y, z_1/u_1, \ldots, z_k/u_k] * \mathsf{ls}_{\Lambda(x,y)}(z, \zeta) * [M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'}{\varphi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon, \zeta) * [M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'}\ C$$

The following rule allows one to remove from the LHS a list segment that forms an initial part of a list segment that appears on the RHS. The condition C requires that $\varphi \models \varepsilon = \varepsilon' \ * \ \mathit{true}$ and that $\Lambda(x,y) \models \Lambda'(x,y)$[14].

[14] We note that this kind of entailment query cannot be discharged in the way we sketched above for the case when the RHS of the entailment is a pure formula (intuitively, one would need some negation over SL). However, such queries can be discharged by a slight modification of the bi-abduction procedure presented in this section – for details see our technical report [22].

$$\texttt{slseg-ls-ls}\frac{\varphi * [M] \ \rhd\ \psi * \mathsf{ls}_{\Lambda'(x,y)}(\zeta, \zeta')}{\varphi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon, \zeta) * [M] \ \rhd\ \psi * \mathsf{ls}_{\Lambda'(x,y)}(\varepsilon', \zeta')}\ C$$

The further rule allows one to remove a possibly empty list segment from the RHS. A corresponding rule for list segments of the LHS is only needed for entailment checking (cf. [22]).

$$\texttt{slseg-remove-right}\frac{\varphi * [M] \ \rhd\ \psi}{\varphi * [M] \ \rhd\ \psi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon, \zeta)}\ \varphi \models \varepsilon = \zeta \ * \ \mathit{true}$$

We have similar rules for *doubly-linked lists* as the ones stated above, which we omit here for ease of exposition (we point out that `dllseg-pt-ls-left` and `dllseg-pt-ls-right` come in two versions because a doubly-linked list can be unrolled from the left as well as from the right).

Next, we state a rule that allows one to *finish* the abduction process.

$$\texttt{learn-finish}\frac{}{\varphi * [\psi] \ \rhd\ \psi * \mathit{true}}\ \varphi * \psi \text{ is satisfiable}$$

The side condition "$\varphi * \psi$ is satisfiable" is intended to ensure that the abduction solution ψ does not lead to useless contracts: a contract $\{\varphi * \psi\}\, f(\cdots)\, \{\cdots\}$ with $\varphi * \psi$ unsatisfiable does not have a configuration that satisfies its pre-condition! Unfortunately, we only have an approximate procedure for checking the satisfiability of symbolic heaps (see our technical report [22]). However, contracts with an unsatisfiable pre-condition are still sound. Hence, we employ the best-effort strategy of using our approximate procedure to prevent as many useless abduction solutions as possible in order to minimize the number of inferred contracts.

Finally, we state two rules of "last resort" that involve quite some guessing and hence can mislead the abduction process and make it fail (or lead to its exponential explosion when all possible variants of applying the rules are attempted). Intuitively, they allow one to *claim equal* fields whose *equality* is not known, but whose *disequality* is not known either (moreover, in the weaker case, one also checks that it can be shown that the fields lie within the same memory block).

$$\texttt{alias-weak}\frac{\varphi * \chi(\varepsilon) * \varepsilon = \varepsilon' * [M] \ \rhd\ \psi * \chi'(\varepsilon')}{\varphi * \chi(\varepsilon) * [\varepsilon = \varepsilon' * M] \ \rhd\ \psi * \chi'(\varepsilon')}\ C_1$$

$$\texttt{alias-strong}\frac{\varphi * \chi(\varepsilon) * \varepsilon = \varepsilon' * [M] \ \rhd\ \psi * \chi'(\varepsilon')}{\varphi * \chi(\varepsilon) * [\varepsilon = \varepsilon' * M] \ \rhd\ \psi * \chi'(\varepsilon')}\ C_2$$

In the rules, $\chi(x)$ and $\chi'(x)$ are any predicates of the form $x \mapsto _$, $\mathsf{ls}_(x, _)$, $\mathsf{dls}_(x, _, _, _)$, or $\mathsf{dls}_(_, _, x, _)$. Further, C_1 is the condition that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \mathit{true}$ and that *not* $\varphi \models \varepsilon \neq \varepsilon' \ * \ \mathit{true}$. On the other hand, C_2 requires that not $\varphi \models \varepsilon \neq \varepsilon' \ * \ \mathit{true}$ only.

The *alias-weak/strong* rules are used in the following situations:

- There is *no other applicable rule*. Instead of failing due to the impossibility of applying other rules, we try to introduce an alias (if possible, by the *alias-weak* rule) and continue with the abduction using the *match*, *split*, or *slseg/dllseg* rules.
- We wish to infer *multiple abduction solutions*. In such a case, whenever *learn-finish* is applicable, we use it to derive one abduction solution, record it, revert *learn-finish*, and then try to derive other solutions by applying an *alias* rule, followed by applying the other rules again.

We now state the correctness of the abduction procedure:

▶ **Theorem 2.** *Let M be any solution computed by the abduction rules, i.e., we have $\varphi * [M] \rhd \psi$. Then, $\varphi * M \models \psi$.*

Proof. We prove the property by induction on the number of rule applications. We observe that the claim holds for the axiom, i.e., the rule `learn-finish`). We further note that, for all non-axiomatic rules of the shape

$$\text{rule-name}\,\frac{\varphi' * [M'] \ \rhd \ \psi'}{\varphi * [M] \ \rhd \ \psi}\ C,$$

we have that $\varphi' * M' \models \psi'$ implies $\varphi * M \models \psi$ (under the condition C). Hence, the claim holds. ◀

Moreover, we observe that the antiframe M is guaranteed to be a quantifier-free symbolic heap in case the input φ to the abduction procedure is a quantifier-free symbolic heap (the abduction rules maintain this shape of φ).

▶ **Example 3.** We consider the abduction problem

$$X \mapsto a * X + 8 \mapsto z * [?] \models Y \mapsto u * Y + 8 \mapsto w * u \mapsto v * X = Y * \mathit{true}.$$

Its solution by our abduction rules looks as follows:

$$\texttt{learn-pure}\,\dfrac{\texttt{match}\,\dfrac{\texttt{match}\,\dfrac{\texttt{learn-finish}\,\dfrac{}{X = Y * a = u * z = w * [u \mapsto v] \ \rhd \ u \mapsto v * \mathit{true}}}{X + 8 \mapsto z * X = Y * a = u * [z = w * u \mapsto v] \ \rhd \ Y + 8 \mapsto w * u \mapsto v * \mathit{true}}}{X \mapsto a * X + 8 \mapsto z * X = Y * [a = u * z = w * u \mapsto v] \ \rhd \ Y \mapsto u * Y + 8 \mapsto w * u \mapsto v * \mathit{true}}}{X \mapsto a * X + 8 \mapsto z * [X = Y * a = u * z = w * u \mapsto v] \ \rhd \ Y \mapsto u * Y + 8 \mapsto w * u \mapsto v * X = Y * \mathit{true}}$$

8 Implementation and Experimental Evaluation

We have implemented the proposed techniques in a prototype tool called Broom. Its source code is publicly available[15] under GNU GPLv3. The tool itself is implemented in OCaml. The SMT queries produced by the tool are answered using the Z3 solver [12]. The front-end of Broom is based on Code Listener [15], a framework providing access to the intermediate code of a compiler (as, e.g., `gcc`).

Our approach requires one to answer entailment queries $\varphi_1 \models \varphi_2$ at several points. If φ_2 is pure, we translate $\varphi_1 \wedge \neg\varphi_2$ from SL into the bitvector theory and ask the underlying SMT solver. However, this cannot be easily done when φ_2 contains a spatial predicate (our fragment of SL is not closed under negation). While it might be possible to develop a general (sound and complete) entailment procedure, e.g., extending [26], we decided to use an approximation based on similar principles as our bi-abduction procedure. We give details of the translation of SL to the bitvector theory as well as of our more general approximated entailment procedure in our technical report [22].

We note that, in the implementation of Broom, we relaxed the requirement put on the *ptrplus* operation of our minilanguage (Sec. 4), which requires that the pointer resulting from the expression $y + z$ stays within the allocated block – i.e., $\mathfrak{b}(S_B(y)) \leqslant S_B(y) + S_B(z) \leqslant \mathfrak{e}(S_B(y))$. According to the C standard, the relaxation of this condition leads to

[15] `https://pajda.fit.vutbr.cz/rogalew/broom`

undefined behaviour, but it is often used in low-level system code as, e.g., in the Linux list implementation. In our implementation, we allow pointers to have values outside of the allocated blocks, but we explicitly track their *provenance* (i.e., the basis wrt which they are defined) using the $\mathfrak{b}$ predicate.

Broom comes with a number of parameters that can be set for the analysis, with the most important being the following ones:

- *Solver timeouts:* Timeouts of the underlying solver can be set separately for symbolic execution, widening, and formula simplification. Using a timeout, one can balance between speed and precision. With a lower timeout, the analysis is faster, but some functions need not be fully analysed due to an abduction or widening failure. The default timeouts used in our later presented experiments are 2000ms for the symbolic execution, 200ms for widening, and 100ms for formula simplification.
- *Number of loop unfoldings:* A limit on the number of loop unfoldings is used to stop the loop analysis when a fixpoint is not computed within a given number of loop iterations. Then, either no contract or partial contracts are returned. The default value used in our experiments is 5.
- *Abduction strategy:* The abduction strategy can be set as follows: In the standard configuration, it follows the order of rules presented in Sec. 7. The tool also supports an alternative strategy where the `alias-weak/strong` rules are used to derive multiple abduction solutions as discussed in Sect. 7. This may lead to an exponential blow-up in the number of contracts for particular functions (a lot of them useless) together with a blowup of the running time. On the other hand, this strategy allows us to fully verify some of our most complicated code fragments (namely, the intrusive lists discussed below). As a part of our future research, we would like to study some heuristically-driven application of this strategy that would not explore so many useless contracts.

Finally, we would like to stress that Broom is now in a stage of a very early prototype, intended mainly to illustrate the theoretical potential of our technique, with huge space for performance optimizations. As a primary source of possible optimisations, we see the way how Broom interacts with the SMT solver (the cost of SMT queries represents a very significant part of the cost of the entire analysis). One way that we see as highly promising for optimisations in this direction is to use static pre-evaluation of some SMT queries – if one can statically evaluate a query, an expensive solver call can be avoided. This can significantly limit the number of SMT queries and improve the running time. We have already partially implemented some static pre-evaluation for the $\varphi \models \varepsilon = \varepsilon' * true$ queries within `match/split` abduction rules, which alone reduced the running time by 25 % at some examples. Further optimization possibilities then lie, e.g., in incremental solving, caching solver results, and/or introducing heuristics to decrease the amount of nondeterminism in the abduction rules. As for the last mentioned possibility, especially in the case of the `match/split` rules there can be several candidate predicates $\varepsilon \mapsto \zeta$ on the LHS and several candidate predicates $\varepsilon' \mapsto \zeta'$ on the RHS, which one needs to consider, and it would be very helpful to have some guidance in this process.

8.1 Experiments

We evaluate our tool Broom on a set of experiments in which we analyse various fragments of list manipulating code. Since Broom is in a highly prototypical stage, we do not venture into analysing large code bases. Instead, we concentrate on shorter but complex code highlighting what our approach implemented in the tool can handle (and what other tools do typically

not manage).

The considered code was pre-processed in the following ways: (1) All appearances of the so-far unsupported constructions `&var` and `var.next` were replaced by `p_var` and `p_var->next`, respectively, where `p_var = alloca(sizeof(*p_var))`. (2) We replaced `for` loops with integer bounds by non-deterministic `while` loops because our abstraction and entailment are currently very limited when working with integers. Both of the above is planned to be resolved within our future work. Further, we analysed all the code assuming that heap allocation always succeeds.

The experiments were run on a machine with an Intel i7-4770 processor with 32 GiB of memory. The current implementation of Broom uses a single core only. We compare our results with those of Infer v1.1.0[16] and Gililan (PLDI'20 version)[17], which are the only tools we are aware of that can analyse at least some of the code we are interested in. We note that Infer was running with debug information enabled (using the command `infer run --debug`) as we wanted to manually check the obtained contracts. The debug option may increase the running time of Infer, but, as one can see in Table 1, the running times are not an issue for Infer.

Table 1 presents a comparison of the results obtained using Broom, Infer, and Gillian on our collection of list-manipulating code fragments.[18] To get the results, Broom was used with its standard abduction strategy where the `alias-weak`/`alias-strong` rules are used only if no other rule is applicable. For each of the cases, the table gives first the total number of functions that the benchmark consists of. Next to it, separately for Broom, Infer, and Gillian, we give the time the tools took for the analysis. Further, we list the number of functions for which the respective tool produced a non-trivial contract. There are up to three numbers in the form $a/b/c$ (b or c can be omitted), representing the number of functions for which the respective tool computes (a) complete contracts, (b) sound but only partial contracts, and (c) error contracts – i.e., preconditions under which a given function is bound to fail, which are provided by Gillian only. Finally, we also provide a remark whether the tool reported some error (or whether it itself hit some internal error). The expected and really obtained analysis results are encoded as follows (including internal errors of an analyser): OK= *no error found*[19], DF= *double free*, ML= *memory leak*, IE= *internal error*, PE= *internal parsing error*.

We now discuss the individual cases in more detail – when doing so, we concentrate on comparing the results of Broom with those of Infer that can get somewhat closer to the results of Broom:

- `circ-DLL`: This example deals with a simple implementation of *circular doubly-linked lists* (whose part is, in fact, used as the running example in Fig. 1). The code includes functions for inserting the first element, inserting another element after an existing one, and for removing elements. Apart from that there is a higher-level function that inserts the first element, the second element, and them removes one of them.[20] The code contains

[16] `https://github.com/facebook/infer/releases/tag/v1.1.0`
[17] `https://github.com/GillianPlatform/Gillian/releases/tag/PLDI20`
[18] All the code is available together with our tool.
[19] We note that, as far as our experience reaches, Gillian produces its error contracts whenever there is a risk of a null-pointer dereference. In many cases, e.g., in the Linux list library, the error summaries provide a correct result, which, however, does not take into account the fact that the library is designed such that the appropriate functions are never called with a null argument. At the same time, Gillian may miss real, higher-level errors present in the code, which were those we expected to be reported. In such cases, we say in the table in the column for obtained results that the (expected) error was not found.
[20] This function can be viewed as an analysis harness while we were stressing that our analysis does not

Table 1 Experiments with the standard abduction strategy of Broom and a comparison with results obtained from Infer and Gillian.

Name	Exp. result	Fncs total	Broom			Infer			Gillian		
			T [s]	Fncs contr	Res	T [s]	Fncs contr	Res	T [s]	Fncs contr	Res
circ-DLL	ML	4	6	4	ML	0.5	1/1	IE	1.2	1/0/2	OK
circ-DLL-err	DF	4	6	4	DF	0.5	1/1	IE	1.2	1/0/2	OK
circ-DLL-embedded	OK	4	9	4	OK	0.5	1/2	IE	0.6	0	PE
Linux-list-1	ML	11	56	10	ML	1.5	2/3	OK	0.6	0	PE
Linux-list-2	OK	11	42	11	OK	0.7	1/6	IE	0.6	0	PE
Linux-list-2-err	ML	11	28	11	ML	0.6	1/6	IE	0.6	0	PE
Linux-list-all	OK	23	267	21/2	OK	1.0	7/15	IE	44	8/0/9	OK
intrusive-list	OK	15	99	10/5	OK	0.7	4/3	OK	0.6	0	PE
intrusive-list-min	OK	9	45	6/2	OK	0.7	1/3	IE	0.6	0	PE
intrusive-list-smoke	OK	20	133	10/5	OK	0.9	4/3	OK	0.6	0	PE

no pointer arithmetic nor any other advanced features. It is intended to show that even in such a case our abduction rules restated wrt [6, 7] can bring some advantage. Namely, this is a consequence of that our use of the per-field separation allows us to cover more shapes of the data structures within a single contract. Indeed, as discussed already in Sect. 2, it produces a single contract for insertion into a cyclic list with one element and with more elements. Infer cannot use the same reasoning and since it primarily favours scalability, it will come with a contract for inserting into lists with at least two elements. Consequently, it then fails to analyse the top level function. As for the memory leak reported by Broom, it is a real error caused by that one of the introduced elements is not deleted.

- `circ-DLL-err` is a variation on `circ-DLL` into which we introduced a double-free error.
- `circ-DLL-embedded` is another variation on `circ-DLL` in which the list implementation from `circ-DLL` is used as a basis of a simple intrusive list in which the list structure with the linking fields from `circ-DLL` is nested into a larger data structure.
- `Linux-list-1` is our first experiment with intrusive lists in the form they are used in the Linux kernel (for some more impression about Linux lists, see Fig. 5). This particular code comes in particular from the benchmark suite of the Predator analyser [16].[21] The code contains multiple different functions for initialisation of the lists, for inserting into it, and for traversing the lists. The top-level code that is present then creates a circular Linux list nested into another circular Linux list. As can be from Fig. 5, the code involves pointer arithmetic (even in a form not supported by the C standard), and the use of nested structures leads to an application of our block splitting rules. The only function that Broom fails to handle is the function for traversing the entire list – the reason is that our so-far quite simple implementation of list abstraction fails in this case, and the otherwise correct computation diverges (which we, however, believe to be solvable in the

need such a harness. Here, we would like to stress that this indeed holds – none of the considered tools needs (nor in any way uses) the top-level function to be able to analyze the other functions. We use the harness as a model of any higher-level code using the list. Moreover, it allows us to show that the contracts that got generated for the particular functions are not complete enough, which shows up in the inability of the appropriate tool to analyse the higher-level functions.

[21] We note here that Predator can analyse the code, but – unlike Broom, Infer, or Gillian – it entirely relies on that the code is closed, i.e., it comes with a main function and has no further inputs.

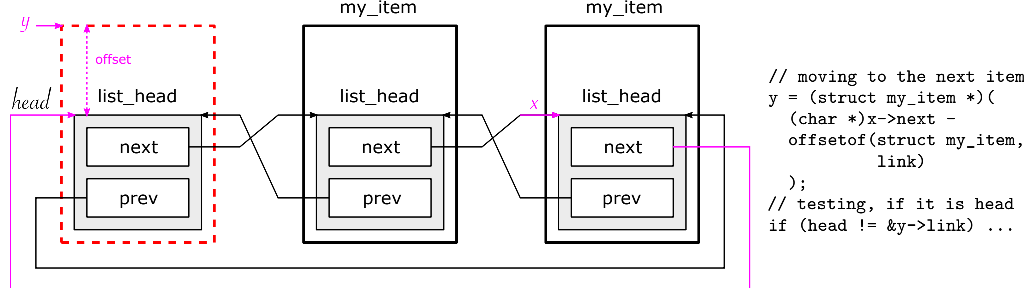


Figure 5 An illustration of the Linux list data structure. The gray boxes represent the linking structure `list_head` that is nested into a user-defined structure `my_item`, whose instances the user needs to be linked into a list. List-manipulating functions know nothing about the user-defined structures: they work with the linking structures only. The user data are accessible through pointer arithmetic only. Note that the head node of the list does *not* have the user-defined envelope. The code shown on the right illustrates how the list is traversed. Note also that when one passes from the last element pointed by `x` to its successor (hence back to the head), the involved pointer arithmetic causes that the pointer `y` will be pointing out of the allocated space, which is, however, correct since it will never be dereferenced (just used for further pointer arithmetic).

close future – indeed, we can get fully inspired by the abstraction used in Predator; the concrete list abstraction used is not specific for our approach). The memory leak reported is a real one – it comes from the top-level function that does not destroy the list.

- `Linux-list-2` is a variation on the above case. It contains functions for an initialisation of a Linux list, inserting elements at its tail, and for deleting the elements. The top-level function initializes the list, inserts several elements, traverses the elements one by one, and deletes them.
- `Linux-list-2-err` is a variation on `Linux-list-2` where one of the inserted elements is not deleted and hence a memory leak is caused.
- `Linux-list-all` contains the entire collection of functions defined for working with Linux lists without any top-level function. The collection includes functions for different kinds of insertion of elements, removal of elements, swapping of elements (both within a list and between lists), moving to the end or to another list, rotation, splicing, etc. We can see that Broom produced complete contracts for many more of the functions. The contracts from Infer often do not cover cases of lists of length 0 or 1. In one of the remaining cases, Infer produced no result; and for the last one, it produced a partial result (that appears not to cover one of the branches of the function).
- `intrusive-list` is the intrusive list library[22]. See Fig. 6 for an illustration how the data structure and the code looks like. Apart from features seen already above (pointer arithmetic and a need to deal with linking fields embedded into larger structures with a need to apply block splitting), the code contains also *bit-masking*. In particular, one bit of the next pointers is used to mark pointers back to the head node, thus effectively marking the "end" of the circular list. A further intricacy of the code is that the insertion into the list touches three nodes that may be different but that may also collapse into a single node. In the case of the Linux list, we have mentioned a similar situation but with two nodes only. Having three nodes that possibly collapse is not only beyond the

[22] Described in the Patrick Wyatt's blog post "Avoiding game crashes related to linked lists", `http://www.codeofhonor.com/blog/avoiding-game-crashes-related-to-linked-lists`, on September 9th, 2012, and implemented in `https://github.com/robbiev/coh-linkedlist`.

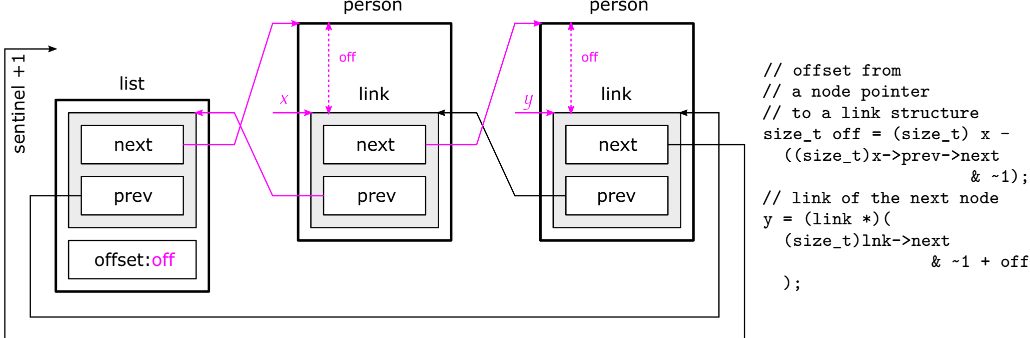


Figure 6 An illustration of the intrusive list data structure. The code fragment shown to the right of the figure gives the code used in the function `link_get_next` to obtain the linking structure of the next node. Note the use of the pointer arithmetic including bit-masking (to clear the bit whose bit-masking on the next pointers is used to denote the sentinel node of the list).

capabilities of Infer but also Broom if it is used with its basic abduction strategy. This is the reason why some of the contracts produced by Broom are also not complete in this case (e.g., effectively allowing insertion into a list with more than one node only). We will, however, show below that Broom can resolve the problem when using more power of the `alias-weak`/`alias-strong` rules, though for the price of quite increased runtime requirements. As for Infer, it is clearly visible that its coverage of the functions is much weaker (interestingly, we noticed that it completely ignored the bit-masking when deriving some of the contracts).

- `intrusive-list-min` contains a subset of the functions considered above (for initializing a list, inserting an element, removing an element) together with a top-level function utilising these functions. Essentially, the intention here was to create an as small as possible example of the given kind already problematic for Infer. Again, even Broom cannot handle it fully under its standard abduction strategy.
- `intrusive-list-smoke` contains the entire intrusive list library from above together with several top-level functions provided by the author to test the library. The tests use structures modelling some personal records to be linked into a list via the embedded linking structures. They create a few such records, link them into a list, traverse them (forward/backward), and destroy the list.

We now proceed to our experiments with the `alias-weak`/`alias-strong` rules. As we have said above, these rules involve a lot of guessing. Hence, if they are used to explore various possible abduction solutions based on different aliasing scenarios, the running time may grow considerably, but it may resolve situations that are otherwise not resolved. To confirm this, we have applied Broom with the strategy of using the `alias-weak`/`alias-strong` rules to explore different possible abduction solutions with different possible aliasing scenarios on the intrusive list case study. The results are shown in Table 2. The first row concerns the experiment `intrusive-list-min` discussed already above. At that time, we noted that Broom could not fully handle some of the intrusive list functions since they required it to merge three possibly independent nodes into a single one. As can be seen in Table 2, with the help of the `alias-weak`/`alias-strong` rules, Broom does fully manage even this problem (though the runtime grew a lot). The next two rows – `intrusive-list-min-2` and `intrusive-list-min-3` – are variations on the previous case where we intentionally introduced some bugs, which were correctly discovered. Finally, the last row shows a

Table 2 Experiments with `alias-weak`/`alias-strong` in Broom.

Name	Expected result	Fncs total	T [m]	Funcs contr	Res
intrusive-list-min	no error	9	46	9	no error found
intrusive-list-min-2	memory leak	9	47	9	memory leak
intrusive-list-min-3	double free	9	49	9	double free
intrusive-list-smoke	no error	20	505	16	no error found

significant improvement even for the entire library of intrusive lists together with its "smoke" tests.

To sum up, we believe that, despite the highly prototypical nature of Broom, the presented experiments show that the proposed approach is indeed capable of handling code that is beyond the capabilities of other currently existing approaches.

9 Conclusion and Future Works

We have presented a new SL-based bi-abduction analysis capable of analysing fragments of code that manipulates with various forms of dynamic linked lists implemented using advanced low-level pointer operations. This includes operations such as pointer arithmetic, bit-masking on pointers, block operations, dealing with blocks of in-advance-unknown size, splitting them into fields of not-fixed size, which can then be merged again, etc. Although our approach builds on a body of previous research, especially, [2, 6, 7, 16], it extends it significantly to handle the mentioned features. In particular, to be able to handle the considered kind of code, we build on a flavor of SL that uses a per-field separating conjunction instead of a per-object separating conjunction, and we also introduce a number of new abduction rules that allow us to deal with pointer arithmetic, block splitting and merging, and so on. We have implemented the proposed approach in a prototype tool Broom. Despite Broom is a very early prototype, our experiments with it allowed us to handle code fragments that are – to the best of our knowledge – out of the capabilities of currently existing analysers.

We believe that there is a lot of space for further improvements of our results in the future. First, we would like to significantly optimize Broom to make it applicable to larger code bases. Here, we are thinking of applying many of the low-level optimisations applied in other tools of a similar kind (replacing as many as possible of SMT queries by answering them using simple static rules, using incremental SMT solving, caching as much information as possible, etc.). Next, we would like to explore possibilities how to reduce the amount of non-determinism present in the abduction when the `alias-weak/strong` rules are applied. The goal is to preserve as much as possible of the power of these rules but reduce the cost of applying them. Perhaps, we could rely here partially on some pre-defined heuristics and partially even on some techniques from machine learning, which are now being applied even in SMT solvers and elsewhere. Next, we would like to significantly improve our implementation of list abstractions (inspired, e.g., by [16]) as well as numerical abstractions. Last but not least, we would also like to think of adding support for other classes of dynamic data structures than lists.

References

1 Andrew W. Appel. *Program Logics - for Certified Compilers*. Cambridge University Press, 2014.

2 J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. of CAV'07*, volume 4590 of *LNCS*. Springer, 2007.

3 A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data. In *Proc. of ATVA'12*, volume 7561 of *LNCS*. Springer, 2012.

4 C. Calcagno and D. Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proc. of NFM'11*, volume 6617 of *LNCS*. Springer, 2011.

5 C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

6 C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *Proc. of POPL'09*. ACM, 2009.

7 C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM*, 58(6), 2011.

8 Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

9 B.-Y.E. Chang, X. Rival, and G.C. Necula. Shape Analysis with Structural Invariant Checkers. In *Proc. of SAS'07*, volume 4634 of *LNCS*. Springer, 2007.

10 W.-N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. *Science of Computer Programming*, 77(9), 2012.

11 C. Curry, Q. Loc Le, and S. Qin. Bi-Abductive Inference for Shape and Ordering Properties. In *Proc. of ICECCS'19*. IEEE, 2019.

12 L.M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.

13 Stéphane Demri, Étienne Lozes, and Alessio Mansutti. The effects of adding reachability predicates in propositional separation logic. In *FoSSaCS*, volume 10803 of *Lecture Notes in Computer Science*, pages 476–493. Springer, 2018.

14 C. Drăgoi, C. Enea, and M. Sighireanu. Local Shape Analysis for Overlaid Data Structures. In *Proc. of SAS'13*, volume 7935 of *LNCS*. Springer, 2013.

15 K. Dudka, P. Peringer, and T. Vojnar. An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proc. of EUROCAST'11*, volume 6927 of *LNCS*. Springer, 2011.

16 K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*, volume 7935 of *LNCS*. Springer, 2013.

17 M. Echenim, R. Iosif, and N. Peltier. Unifying Decidable Entailments in Separation Logic with Inductive Definitions. In *Proc. of CADE'21*, volume 12699 of *LNCS*. Springer, 2021.

18 C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. Compositional Entailment Checking for a Fragment of Separation Logic. In *Proc. of APLAS'14*, volume 8858 of *LNCS*. Springer, 2014.

19 B. Fang and M. Sighireanu. Hierarchical Shape Abstraction for Analysis of Free List Memory Allocators. In *Proc. of LOPSTR'16*, volume 10184 of *LNCS*. Springer, 2016.

20 J. Heinen, T. Noll, and S. Rieger. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. In *Proc. of TSS'09*, volume 266 of *ENTCS*. Elsevier, 2010.

21 L. Holík, O. Lengál, J. Šimáček, A. Rogalewicz, and T. Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, volume 8044 of *LNCS*. Springer, 2013.

22 L. Holík, P. Peringer, A. Rogalewicz, V. Šoková, T. Vojnar, and F. Zuleger. Low-Level Bi-Abduction, 2022. `arXiv:2205.02590`.

23 R. Iosif, A. Rogalewicz, and T. Vojnar. Deciding Entailments in Inductive Separation Logic with Tree Automata. In *Proc. of ATVA'14*, volume 8837 of *LNCS*. Springer, 2014.

24 S. Ishtiaq and P.W. O'Hearn. Separation and Information Hiding. In *Proc. of POPL'01*. ACM, 2001.

25 J.L. Jensen, M.E. Jørgensen, M.I. Schwartzbach, and N. Klarlund. Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In *Proc. of PLDI'97*. ACM, 1997.

26 J. Katelaan and F. Zuleger. Beyond Symbolic Heaps: Deciding Separation Logic With Inductive Definitions. In *Proc. of LPAR'11*, volume 73 of *EPiC Series in Computing*. EasyChair, 2020.

27 Q. Loc Le. Compositional Satisfiability Solving in Separation Logic. In *Proc. of VMCAI'21*, volume 12597 of *LNCS*. Springer, 2021.

28 Q. Loc Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape Analysis via Second-Order Bi-Abduction. In *Proc. of CAV'14*, volume 8559 of *LNCS*. Springer, 2014.

29 P. Maksimovic, S.-É. Ayoun, J.F. Santos, and P. Gardner. Gillian, Part II: Real-World Verification for JavaScript and C. In *Proc. of CAV'21*, volume 12760 of *LNCS*. Springer, 2021.

30 P. Maksimovic, J.F. Santos, S.-É. Ayoun, and P. Gardner. Gillian: A Multi-Language Platform for Unified Symbolic Analysis, 2021. `arXiv:2105.14769`.

31 V. Malik, M. Hruška, P. Schrammel, and T. Vojnar. Template-Based Verification of Heap-Manipulating Programs. In *Proc. of FMCAD'18*. IEEE, 2018.

32 Jens Pagel and Florian Zuleger. Strong-separation logic. In *ESOP*, volume 12648 of *Lecture Notes in Computer Science*, pages 664–692. Springer, 2021.

33 J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE, 2002.

34 M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3), 2002.

35 J.F. Santos, P. Maksimovic, S.-É. Ayoun, and P. Gardner. Gillian: Compositional Symbolic Execution for All, 2020. `arXiv:2001.05059`.

36 J.F. Santos, P. Maksimovic, S.-É. Ayoun, and P. Gardner. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *Proc. of PLDI'20*. ACM, 2020.

37 T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On Verifying Complex Properties using Symbolic Shape Analysis. In *Proc. of HAV'07*, 2007.

38 H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, volume 5123 of *LNCS*. Springer, 2008.

39 K. Zee, V. Kuncak, and M.C. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI'08*. ACM, 2008.

Functional Programming for Distributed Systems with XC

Giorgio Audrito
University of Turin, Italy

Roberto Casadei
University of Bologna, Cesena, Italy

Ferruccio Damiani
University of Turin, Italy

Guido Salvaneschi
Universität St. Gallen, Switzerland

Mirko Viroli
University of Bologna, Cesena, Italy

Abstract

Programming distributed systems is notoriously hard due to – among the others – concurrency, asynchronous execution, message loss, and device failures. *Homogeneous* distributed systems consist of similar devices that communicate to neighbours and execute the same program: they include wireless sensor networks, network hardware, and robot swarms. For the homogeneous case, we investigate an experimental language design that aims to push the abstraction boundaries farther, compared to existing approaches.

In this paper, we introduce the design of XC, a programming language to develop homogeneous distributed systems. In XC, developers define the single program that every device executes and the overall behaviour is achieved collectively, in an emergent way. The programming framework abstracts over concurrency, asynchronous execution, message loss, and device failures. We propose a minimalistic design, which features a single declarative primitive for communication, state management, and connection management. A mechanism called *alignment* enables developers to abstract over asynchronous execution while still retaining composability. We define syntax and operational semantics of a core calculus, and briefly discuss its main properties. XC comes with two DSL implementations: a DSL in Scala and one in C++. An evaluation based on smart-city monitoring demonstrates XC in a realistic application.

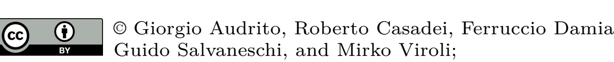
2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Functional constructs; Theory of computation → Operational semantics; Theory of computation → Type structures; Computing methodologies → Distributed programming languages

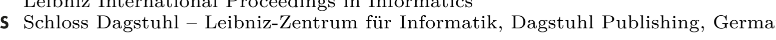
Keywords and phrases Core calculus, operational semantics, type soundness, Scala DSL

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.20

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.8`
Software (XC/Scala DSL): `https://github.com/scafi/artifact-2021-ecoop-xc`
archived at `swh:1:dir:b8f42b1ff5725d1af15c4ee5fce324e6cd54da4a`
Software (XC/Scala SmartC case study): `https://github.com/scafi/artifact-2021-ecoop-smartc`
archived at `swh:1:dir:1eb857ef9c19996a73bdd2ceb61c583b953b42b7`

Funding This work was supported by the EU/MUR FSE REACT-EU PON R&I 2014-2022 (CCI2014IT16M2OP005), the Swiss National Science Foundation (SNSF, No. 200429), the Hessian LOEWE initiative emergenCITY, and the Ateneo/CSP "Bando ex post 2020".


© Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 20; pp. 20:1–20:28

LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Programming distributed systems is notoriously hard because they require reasoning about a number of issues that inevitably arise in this setting, including concurrency, remote communication, asynchronous execution, message loss and device failures.

The design of programming languages for distributed systems attempts to address these issues by carefully combining cases where *(i)* programmers are given *explicit control* over certain aspects of distribution, *(ii)* the design employs *abstraction* to hide low level mechanisms from developers. A well-known example of explicit control is fault tolerance in the actor model, where developers can define a reaction strategy in case of failures (through the so-called actor supervision hierarchy) [3]. However, the actor model abstracts over placement and – at least in the original actor model – actors communicate with actors on the same machine the same way they do with a remote actor. Similar combinations of abstraction and explicit control position other distributed programming languages in the design space. For example, in the MPI model for HPC, processes are organized in topologies and can explicitly send messages close to them in such topology [45]. In Partitioned Global Address Space Languages [28], the programming model abstracts over the memory separation across processes. Pub-sub systems abstract over the binding between message sender and receivers ensuring that senders and receivers can seamlessly join and leave the system [31]. In summary, the design of these programming models stems from a combination of explicit control ensured to the user and details that are abstracted over.

An important class of distributed systems are *homogeneous* and *situated*, i.e., they are composed of similar devices that communicate with "neighbours", and execute similar programs. This property has been observed, e.g., in distributed systems for hierarchical control of network routing [37], crowd management by handheld devices [16], Wireless Sensor Network (WSN) connectivity management [36] and gossip-based data aggregation [35], task allocation in robot swarms [18, 46], and coordination of enterprise servers [24]. More applications are emerging, pushed by the scientific and technological trends of the Internet of Things (IoT) and of Cyber-Physical Systems (CPS) [47], and of the coordination of mobile agents [40]. Crucially, "homogeneity" in large-scale systems also stems from the case where each device runs a program from a predefined set – corresponding to a homogeneous configuration with a single program with an initial branch.

In this paper, we address the issue of programming such a class of homogeneous systems. Over time, several approaches have been proposed to address these kinds of systems, including *spatial computing* [29], *ensemble-based programming* [27, 1], and, notably, *field-based computing* [49, 38, 40], where the overall distributed system behaviour is understood as producing a *computational field*, i.e., a map from network nodes to values. Inspired by these works, we investigate XC, a novel programming language design that captures their essence (as detailed in Section 7) *into a single construct* aiming to abstract over low-level concerns developers face in distributed systems, while allowing differentiated messages to neighbours. We show a design where concurrency, asynchronicity, network communication, message loss, and failures do not need to be handled explicitly. Thanks to a mechanism that is referred to as *alignment*, distributed programs written in this style retain composability even if devices operate fully independently, waking up, executing the program and communicating asynchronously at arbitrary times and frequencies. Messages from other devices are processed homogeneously, hence developers do not need to separately handle the case when a message is lost or a device fails: such lost message is simply not part of the (homogeneous) computation. All required computational mechanisms can be unified into a single declarative construct called `exchange`, which provides *(i)* access to neighbours' values, *(ii)* persistence of information for subsequent executions, *(iii)* communication with neighbours, and *(iv)* compositional behaviour.

To summarise, this paper provides the following contributions:

- We describe the design of XC, a programming language for homogeneous distributed systems that abstracts over concurrency, network communication, message loss, and device failures. Crucially, XC retains compositionality even with asynchronous communication, thanks to alignment.
- We show that XC can effectively capture a number of applications in distributed systems, including distributed protocols such as gossiping, finding an optimised communication channel, and common applications in self-organizing systems.
- We provide a formalization of a core calculus for XC, including syntax and operational semantics, and briefly discuss its main properties.
- We implement XC as publicly available Scala and C++ internal DSLs, together targeting a number of different execution platforms.
- In addition to the applications above, we evaluate our approach on a case study demonstrating XC's applicability to real-world scenarios and its compositionality, and answering two research questions: *(RQ1)* whether the decentralised execution of the XC program on each device induces the desired *collective* behaviour; and *(RQ2)* to what extent such behaviour can be expressed by composition of simpler functions.

The paper is structured as follows. Section 2 introduces XC design, Section 3 demonstrates XC through examples, Section 4 presents the formalization, Section 5 discusses the implementation, Section 6 evaluates XC, Section 7 compares XC to the related work, Section 8 concludes and outlines future research directions.

2 XC Language Design

2.1 System model

Asynchronous, round-based execution and communication. We consider *devices* that send/receive *messages* with (physical or logical) *neighbours*. The set of neighbours of any device can change dynamically to model, e.g., spatial movement, failures, and network delay. Existing homogeneous systems (cf. [15]) typically work with devices that repeatedly execute a computation aimed at producing a message for some neighbours, whereas message reception is asynchronous. Therefore, we abstract device behaviour through a notion of *(execution) round*, whereby a device independently "fires" and "atomically executes" a XC program, then it sends a resulting message to neighbours before waiting to execute the round again – sometimes we say it "wakes up", execute the round, and then "go back to sleep". The behaviour of each device in the network is developed as a single program[1]. Such execution rounds may occur at comparable periodic intervals on all devices but there is no such assumption in general (every device may have its own scheduling of rounds). Indeed, a device can run out of battery and never wake up again, or it can restart waking up after a long time if the battery gets charged. In summary, rounds – hence the communication among devices – are entirely asynchronous.

Last-message buffering and dropping. The messages received by a sleeping device queue up in a buffer. When the device wakes up, it executes a XC program that processes such messages, producing new messages to send out. Such messages are eventually processed by the neighbours when they wake up for their next round. For example, in the system execution

[1] This approach is often referred to as *macroprogramming* [44] or *multi-tier programming* [50]. It does not restrict realisable behaviours as devices can still exhibit different executions of the same program.

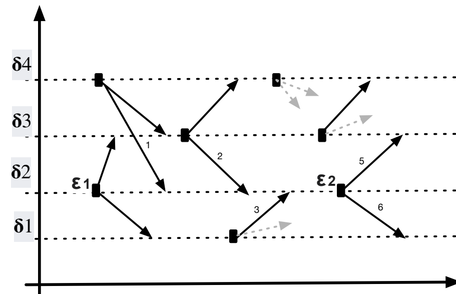


Figure 1 XC system model.

in Figure 1, there are four devices δ_1 to δ_4. In the considered time span, device δ_2 wakes up twice and performs two computation rounds, ϵ_1 and ϵ_2. Grey arrows indicate messages that get lost and are never received. The computation ϵ_2 processes three messages, received from δ_4, δ_3, and δ_1 while δ_2 was asleep. After the computation, δ_2 sends out a message to δ_3 and to δ_1. The order of messages from a same sender is preserved but, other than that, there are very few assumptions on messages. If a device δ_1 runs multiple rounds before a device δ_2 even runs a single round, δ_2 sees only the message received from the last round of δ_1, i.e., newly received messages from a same sender overwrite older ones. Also, messages are not removed from the buffer after reading them, unless they *expire* (i.e., are deemed too old according to any pre-established criterion) or unless they are replaced by a new message from the same device, allowing messages to (possibly) persist across rounds. The XC design abstracts over the specific expiration criteria: common choices include removing messages after each read, or after a validity time elapses. This time interval is highly application-specific and stems from a trade-off between *(i)* tolerance to communication delays and failures, and *(ii)* recovery speed after truthful changes on data and neighbourhoods.

When a device δ wakes up, it usually does not find messages from every other device in the system: *(i)* another device may be too far to send a message to δ; *(ii)* messages may get lost; *(iii)* devices may disappear or fail; *(iv)* a device may reboot, losing its queue of received messages; *(v)* δ may deem messages from some devices to be expired. Crucially, XC does not require distinguishing among those cases. When a device wakes up, it finds some messages from (the most recent available execution round of) some other devices. The devices for which a message is available in a certain round are *the neighbours* for that round.

This system model and the terminology associated to it (e.g., "send message to a neighbour") is adopted throughout the paper. These design choices make XC agnostic to the actual communication channel, topology creation and discovery mechanism: e.g., push or pull, broadcast or point-to-point. For example, the same programming model would apply even if a device, after waking up, contacts the neighbours to fetch their current value in a *pull* fashion. Instead, in a network of micro-controller devices, Bluetooth 5.0 *extended advertisements* could be used to share data with neighbour devices in physical proximity, without an explicit discovery mechanism, as the topology is induced by the messages that are actually received. Such an implementation would also grant *causal consistency* [2]. On the other hand, a network of higher-end devices may communicate point-to-point over IP, with discovery mechanisms based on broadcasted messages or rendezvous servers.

2.2 XC's key data type: Neighbouring Values

Datatypes in XC. XC features two kinds of values. *Local* values ℓ include traditional types A like float, string or list. Neighbouring values (*nvalues*) are a map $\underline{\mathtt{w}}$ from device identifiers δ_i to corresponding local values ℓ_i, with a *default* ℓ, written $\ell[\delta_1 \mapsto \ell_1, \dots, \delta_n \mapsto \ell_n]$. A nvalue is

used to describe the (set of) values received from and sent to neighbours. In highly decoupled distributed systems *only a few* neighbours may *occasionally* produce a value. The devices with an associated entry in the nvalue are hence usually a subset of all devices, e.g., because a device is too far to provide a value or the last provided value has expired. The default is used when a value is not available for some reason as will be discussed later (e.g., if a device just appeared and has not yet produced a value). For this reason, it is convenient to adopt the notation above and read it "the nvalue $\underline{\mathtt{w}}$ is ℓ everywhere (i.e., for all neighbours) except for devices $\delta_1, \dots, \delta_n$ with values $\ell_1, \dots, \ell_n$.

To exemplify nvalues, in Figure 1, upon waking up for computation ϵ_2, δ_2 may process a nvalue $\underline{\mathtt{w}} = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2, \delta_1 \mapsto 3]$, corresponding to the messages carrying the scalar values 1, 2, and 3 received when asleep from δ_4, δ_3, and δ_1. The entries for all other devices default to 0. After the computation, δ_2 may send out the messages represented by the nvalue $\underline{\mathtt{w}}' = 0[\delta_3 \mapsto 5, \delta_1 \mapsto 6]$; so that 5 is sent to δ_3, 6 is sent to δ_1, and 0 is sent to every other device (such as a newly-connected device with no dedicated value yet). We also use the notation $\underline{\mathtt{w}}(\delta')$ for the local value ℓ' if $\delta' \mapsto \ell'$ is in $\underline{\mathtt{w}}$, or the default local value ℓ of $\underline{\mathtt{w}}$ otherwise, reflecting the interpretation of nvalues as maps with a default. For instance, $\underline{\mathtt{w}}'(\delta_1) = 6$ and $\underline{\mathtt{w}}'(\delta_2) = 0$. To help the reader, in code snippets, we underline the variables holding neighbouring values, and, similarly, we underline a primitive type $\underline{A}$ to indicate the type of a nvalue $\underline{\mathtt{w}} = \ell[\delta_1 \mapsto \ell_1, \dots, \delta_n \mapsto \ell_n]$ where $\ell, \ell_1, \dots \ell_n$ have type A.

Nvalues generalize local values. A local value ℓ can be automatically converted to a nvalue $\ell[]$ with a default value for every device. In fact, the distinction between local values and nvalues is only for clarity: local values can be considered equivalent to nvalues where all devices are mapped to a default value. In the formalization (Section 4) local values and nvalues are treated uniformly. Functions on local values are implicitly lifted to nvalues, by applying them on the maps' content pointwise. For example, given $\underline{\mathtt{w}}_1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2]$ and $\underline{\mathtt{w}}_2 = 1[\delta_4 \mapsto 2]$, we have $\underline{\mathtt{w}}_3 = \underline{\mathtt{w}}_1 + \underline{\mathtt{w}}_2 = 1[\delta_4 \mapsto 3, \delta_3 \mapsto 3]$. Note that $\delta_3 \mapsto 3$ in $\underline{\mathtt{w}}_3$ is due to the fact that $\delta_3 \mapsto 2$ in $\underline{\mathtt{w}}_1$ and δ_3 has default value 1 in $\underline{\mathtt{w}}_2$. Using also the automatic promotion of local values to nvalues, we have that $\underline{\mathtt{w}}_1 + 1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2] + 1 = 1[\delta_4 \mapsto 2, \delta_3 \mapsto 3]$.

Operations on nvalues. Besides pointwise manipulation, nvalues can be folded over, similar to a list, through built-in function $\mathtt{nfold}(f : (A, B) \to A, \underline{\mathtt{w}} : \underline{B}, \ell : A) : A$, where the function f is repeatedly applied to neighbours' values in a field $\underline{\mathtt{w}}$ (thus excluding the value for the *self* device), starting from a base local value ℓ. For instance, assume that δ_2 is performing a `nfold` operation, with the current set of neighbours $\{\delta_1, \delta_3\}$. Then $\mathtt{nfold}(+, \underline{\mathtt{w}}_1, 10) = 10 + \underline{\mathtt{w}}_1(\delta_1) + \underline{\mathtt{w}}_1(\delta_3) = 10 + 0 + 2$, where $\underline{\mathtt{w}}_1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2]$ is as above. As nvalues should be agnostic to the ordering of the elements (i.e., the ordering of the identifiers δ'), we usually assume that f is associative and commutative.

Sensors and actuators. Since XC programs may express the collective behaviour of homogeneous systems situated in some (physical or computational) environment, the devices are typically equipped with *sensors* and *actuators*. Sensors, in particular, are meant to provide access to contextual and environmental information. These can be accessed by the program through built-in functions as shown in next sections. In a round, similarly to how messages are considered, the program is executed against the most recent sample of sensor values. On the other hand, actuators can be run at the end of the round against the program output (which may collect all the desired actuation commands).

▶ **Example 1** (Distance estimation). A node can estimate its distance from another node in the network by leveraging an existing estimate $\underline{\mathtt{n}}$ provided by its neighbours. To this end, one

Name	Type scheme	Description
Communication:		
`exchange`	$(A, (\underline{A}) \to (T, \underline{A})) \to T$	*Exchanges messages*
Neighbouring value manipulation:		
`nfold`	$((A, B) \to A, \underline{B}, A) \to A$	*Folding of a neighbouring value*
`self`	$(\underline{A}) \to A$	*Extract the self-message*
`updateSelf`	$(\underline{A}, A) \to \underline{A}$	*Update the self-message*
Sensors used in examples:		
`uid`	`num`	*Unique device identifier*
`senseDist`	$\underline{\texttt{num}}$	*Distance estimates to neighbours*
Point-wise operators:		
`+`, `-`, `*`, `/`	$(\texttt{num}, \texttt{num}) \to \texttt{num}$	*Arithmetic operators*
`and`, `or`	$(\texttt{bool}, \texttt{bool}) \to \texttt{bool}$	*Boolean operators*
`==`, `<=`, `>=`	$(A, A) \to \texttt{bool}$	*Relational operators*[2]
`mux`	$(\texttt{bool}, A, A) \to A$	*Multiplexer operator*
`pair`	$(A, B) \to (A, B)$	*Pair creation*
`fst`	$((A, B)) \to A$	*First element of a pair*
`snd`	$((A, B)) \to B$	*Second element of a pair*
Constructors:		
`-1`, `0`, `0.25`, `1`, `Infinity`	`num`	*Numeric constructors*
`True`, `False`	`bool`	*Boolean constructors*
`Pair`	$(A, B) \to (A, B)$	*Pair constructor*

Figure 2 XC: name, type scheme and description of built-in data constructors and functions.

selects the minimum (using `nfold` with starting value `Infinity`) of neighbours' estimates $\underline{\texttt{n}}$ increased by the relative distance estimates $\underline{\texttt{senseDist}}$ (provided by a sensor in the device).

```
1 def distanceEstimate(n) { // has type scheme: (num) → num
2   nfold(min, n + senseDist, Infinity)
3 }
```

Notice that $\underline{\texttt{n}}$ and $\underline{\texttt{senseDist}}$ sum up neighbour-wise; if neighbour δ shares estimate $\underline{\texttt{n}}(\delta)$, the node's best estimate from that neighbour is $\underline{\texttt{n}}(\delta) + \underline{\texttt{senseDist}}(\delta)$. The minimum among all estimates is selected, or `Infinity` if no neighbour is available. ⌟

Additional built-in operations on nvalues are $\texttt{self}(\underline{\texttt{w}} : \underline{A}) : A$, which returns the local value $\underline{\texttt{w}}(\delta)$ in $\underline{\texttt{w}}$ for the self device δ, and $\texttt{updateSelf}(\underline{\texttt{w}} : \underline{A}, \ell : A) : \underline{A}$ which returns a nvalue equal to $\underline{\texttt{w}}$ except for the self device δ – updated to ℓ. The *substitution* notation stand for defaulted map updates, so that $\texttt{updateSelf}(\underline{\texttt{w}}, \ell) = \underline{\texttt{w}}[\delta \mapsto \ell]$. Indeed, the notation $\ell[\delta_1 \mapsto \ell_1, ...]$ for nvalues can be understood as a substitution updating ℓ (the map equal to ℓ everywhere) by associating ℓ_n to δ_n.

XC operators on nvalues behave uniformly on neighbours to encourage uniform behaviour on each element of a nvalue. This approach is idiomatic in XC and results in a more resilient behaviour – inherently tolerate changes of neighbourhoods between rounds. Yet, non-uniform behaviour can be encoded via built-in function `uid` (combined with communication primitives, Section 2.3), which provides the unique identifier δ of the current device.

Figure 2 shows a summary of every built-in function used in this paper. Constructors and point-wise operators are standard; the *multiplexer* operator $\texttt{mux}(\ell_1, \ell_2, \ell_3)$ returns ℓ_2 if ℓ_1 is `True`, ℓ_3 otherwise. We also omit `pair` and use the shortcut $(\texttt{v}_1, \texttt{v}_2)$ for pair construction, and use infix notation for binary operators whenever convenient. Built-ins for neighbouring values has just been discussed. We introduce the `exchange` operator in the next section.

2.3 Communication in XC: Exchange

XC features a single communication primitive $\texttt{exchange}(\texttt{e}_i, (\underline{\texttt{n}}) \Rightarrow \texttt{return } \texttt{e}_r \texttt{ send } \texttt{e}_s)$ which de-sugars to $\texttt{exchange}(\texttt{e}_i, (\underline{\texttt{n}}) \Rightarrow (\texttt{e}_r, \texttt{e}_s))$ and is evaluated as follows. *(i)* the device computes the local value ℓ_i of $\texttt{e}_i$ (the *initial* value). *(ii)* it substitutes variable $\underline{\texttt{n}}$ with the nvalue $\underline{\texttt{w}}$ of

[2] The generic A type in relation-based operators is not allowed to be a function type.

messages received from the neighbours for this exchange, using ℓ_i as default. The exchange returns the (neighbouring or local) value $\mathtt{v}_r$ from the evaluation of $\mathtt{e}_r$. *(iii)* $\mathtt{e}_s$ evaluates to a nvalue $\underline{\mathtt{w}}_s$ consisting of local values to be sent to neighbour devices δ', that will use their corresponding $\underline{\mathtt{w}}_s(\delta')$ as soon as they wake up and perform their next execution round.

Often, expressions $\mathtt{e}_r$ and $\mathtt{e}_s$ coincide, hence we provide $\mathtt{exchange}(\mathtt{e}_i, (\underline{\mathtt{n}}) \Rightarrow \mathtt{retsend}\, \mathtt{e})$ as a shorthand for $\mathtt{exchange}(\mathtt{e}_i, (\underline{\mathtt{n}}) \Rightarrow (\mathtt{e}, \mathtt{e}))$. Another common pattern is to access neighbours' values, which we support via $\mathtt{nbr}(\mathtt{e}_i, \mathtt{e}_s) = \mathtt{exchange}(\mathtt{e}_i, (\underline{\mathtt{n}}) \Rightarrow \mathtt{return}\ \underline{\mathtt{n}}\ \mathtt{send}\ \mathtt{e}_s)$. In $\mathtt{nbr}(\mathtt{e}_i, \mathtt{e}_s)$, the value of expression $\mathtt{e}_s$ is sent to neighbours, and the values received from them (gathered in $\underline{\mathtt{n}}$ together with the default from $\mathtt{e}_i$) are returned as a nvalue, thus providing a view on neighbours' values of $\mathtt{e}_s$.

It is crucial for the expressivity of XC that *exchange* (hence *nbr*) can send a different value to each neighbour, to allow custom interaction, as exemplified below. Next, we show the *self-organising distance* algorithm which showcases the interplay of *exchange* and *nfold*.

▶ **Example 2** (Ping-pong counter). The following function produces a neighbouring value of "connection counters" with neighbours, i.e., it associates every neighbour to the number of times a mutual connection has been established with it.

```
1 def ping-pong() { // has type scheme: () → num
2   exchange( 0, (n) => retsend n + 1 )
3 }
```

Every time a device evaluates `ping-pong`, it first gathers a neighbouring value $\underline{\mathtt{w}}$ associating neighbours to their respective connection counter – 0 is for newly connected devices. Expression $\underline{\mathtt{n}} + 1$ is computed substituting $\underline{\mathtt{w}}$ for $\underline{\mathtt{n}}$, incrementing each such counter (including for newly connected devices, which now map to 1). The resulting value $\underline{\mathtt{w}} + 1$ is both returned by the expression and shared with neighbours. As long as a connection between two devices is maintained, each receives a connection counter from the other and increments it before sending it back – overall counting the messages bouncing back-and-forth. Once a connection between devices breaks and the corresponding messages expire, the connection counter resets to 0, then starts increasing again in case a connection is re-established. Crucially, the program sends different values to neighbours to keep a distinct connection counter with each. ⌟

▶ **Example 3** (Self-organising distance). Computing the minimum distance from any device of the network to a set of *source* devices results in a *gradient* structure [10]. Gradients are a key self-organisation pattern with several applications like estimating long-range distances and providing directions to move data along minimal paths. Function `distanceTo` offers a simple implementation, consisting of a distributed version of the Bellman-Ford algorithm [26].

```
1 def distanceTo(src) { // has type scheme: (bool) → num
2   exchange( Infinity, (n) => retsend mux(src, 0, distanceEstimate(n)) )
3 }
```

Its repeated application in a (possibly mobile) network of devices stabilises to the expected distances from devices where `src` is true. The `exchange` expression in the body updates a local estimate of the distance by *(i)* using `Infinity` as default distance; *(ii)* returning distance zero on source devices; *(iii)* in other devices, selecting the minimum of neighbours' estimates increased by the relative distance estimates (Example 1). If such estimated distance is d, then d is both shared with neighbours (as a constant map with the same estimate d for every neighbour) and returned by the function. Operator `mux` (i.e., a strict version of `if` that computes both its branches, and then selects the output of one of them as result based on the condition) is needed, as sources, though returning 0, must also evaluate

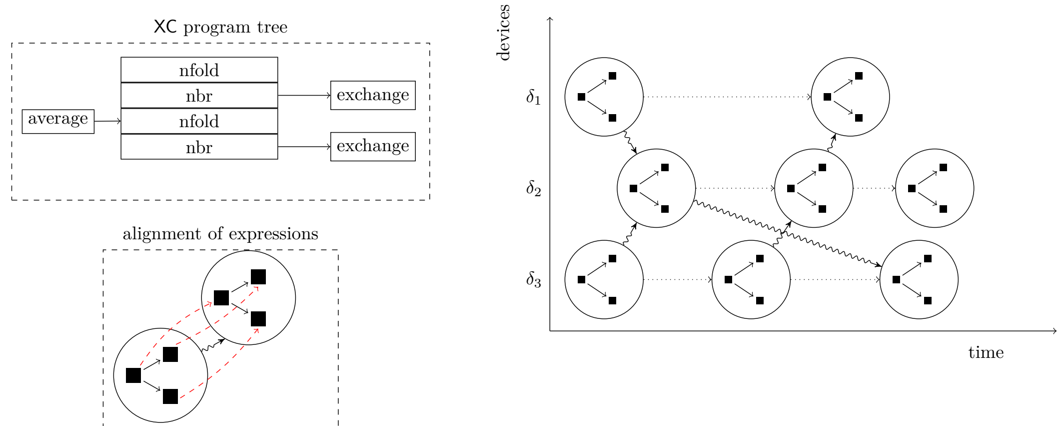


Figure 3 XC alignment mechanism for Example 4.

function call `distanceEstimate` (thus sharing their value $\underline{\mathtt{n}}$). Any change in the network (e.g., due to failure, mobility, dynamic joining) directly affects the domain of $\underline{\mathtt{n}}$, hence the local computation and eventually the whole network – resulting in inherent adaptiveness. ◄

2.4 Compositionality through alignment

If a program executes multiple exchange-expressions, XC ensures through *alignment* that the messages are dispatched to corresponding exchange-expressions across rounds.

► **Example 4** (Neighbour average). The following function `average` computes the weighted average of a value across the immediate neighbours of the current device:

```
1 def average(weight, value) { // has type scheme: (num) → num
2   val totW = nfold(+, nbr(0, weight), weight);
3   val totVl = nfold(+, nbr(0, weight*value), weight*value);
4   totV / totW
5 }
```

First, the total weight of neighbours is computed in Line 2, by first producing a nvalue of neighbours' weights through `nbr(0, weight)`, and then reducing it to its total by `nfold`, using `weight` as base value to ensure that the weight of the current device is also considered. A similar operation is performed in Line 3, where the products `weight*value` of neighbours (including the current device) are added. The weighted average is then obtained as `totV / totW` and returned by the function.

This function contains two calls to `nbr`, which in turn perform calls to the `exchange` built-in, both with messages of type `num`. XC ensures that the messages from the different communicating routines are correctly dispatched to neighbours, each used only in the corresponding call to `exchange` in the neighbours, thus not mixing values and weights. ◄

XC ensures that the values produced by an exchange are processed by the *corresponding* exchange in the next round, i.e., the exchange in the same position in the AST and in the same stack frame. Considering both the AST *and* the stack frame ensures that exchange operations are correctly aligned also in case of branches, function calls and recursion. Figure 3 demonstrates alignment. Top-left is a tree representation of the XC program in Example 4, accounting for stack frames and children in the AST. The larger box with multiple compartments denotes the AST of a function, considering only `exchange`, `nfold`, and functions using

them. Top-right is a system execution. Dotted arrows connect a round (circle) to the next on the same device, and curly arrows denote messages. Within each round we show a tree corresponding to the one top-left. Note that all rounds execute the same tree. Bottom-left zooms into two rounds of different devices evaluating `average` with fully aligned program executions: *corresponding* expressions at the same tree locations interact and consider each other among neighbouring values. Red dashed arrows connecting exchange expressions that belong to different rounds show this interaction. We will discuss partial alignment in the next section, after introducing conditionals. Alignment is a crucial feature in XC because it *enables functional composition of distributed behaviour*, ensuring that messages are transparently dispatched in the correct way, as exemplified in the following.

▶ **Example 5** (Fire detection). Function `closestFire` returns the distance from the closest likely fire (if any), by relying on the simpler functions `average` and `distanceTo`, based on arguments `temperature` and `smoke` which we can assume to be provided by available sensors.

```
1 def closestFire(temperature, smoke) { // has type scheme: (num,num) → num
2   val trust = nfold(+, 1, 1);
3   val hot = average(trust, temperature) > 60;
4   val cloudy = average(trust, smoke) > 10;
5   distanceTo(hot and cloudy)
6 }
```

In Line 2 the function establishes a trust level for the node, which is proportional to the number of neighbours of that node (thus considering central nodes as more relevant), computed as `nfold(+, 1, 1)`. Line 3 checks whether the average temperature, weighted by trust, is above 60 degrees Celsius. Similarly, Line 4 checks whether the average concentration of smoke, also weighted by trust, is above 10%. Finally, Line 5 computes distances to places where both conditions are met (high temperature and smoke) through function `distanceTo`. Several exchange calls are evaluated by both the `average` and `distanceTo` functions: thanks to alignment, the messages processed by each of them are those generated by the same ones in previous rounds of neighbouring devices. ⌟

2.5 Conditionals

XC supports `if (cond) {e1} else {e2}` conditional expressions. Crucially, their semantics interplays with the communication semantics of XC. Since only the exchange operations in the same position within the AST and stack frame align, with a conditional, an exchange aligns *only* across the devices that take the same branch. Thus, while evaluating an XC sub-expression, we consider only *aligned neighbours*, that are round neighbours which evaluated the same sub-expression (as AST and stack frame). Non-aligned neighbours are never considered in the evaluation of the sub-expression, e.g., for the construction of the $\underline{\mathtt{w}}$ of received messages in an `exchange`, or for determining which values of a nvalue should be folded over by a `nfold`. As a result, a conditional expression *splits* the network into two non-communicating sub-networks, each evaluating a different branch without cross-communication.

▶ **Example 6** (Domain-isolated distance computations). Consider a connected network of service requesters and providers. Suppose these nodes are dynamically split into two domains: those involved in local computations (`local`) and those offloading computations (not `local`) to `gateways`, special service providers which provide cloud access. We may want to compute the distance to `gateways` without considering the devices involved in local computations.

```
1 def distanceToGateways(local, gateway) { // has type scheme: (bool,bool) → num
2   if (local) { Infinity } else { distanceTo(gateway) }
3 }
```

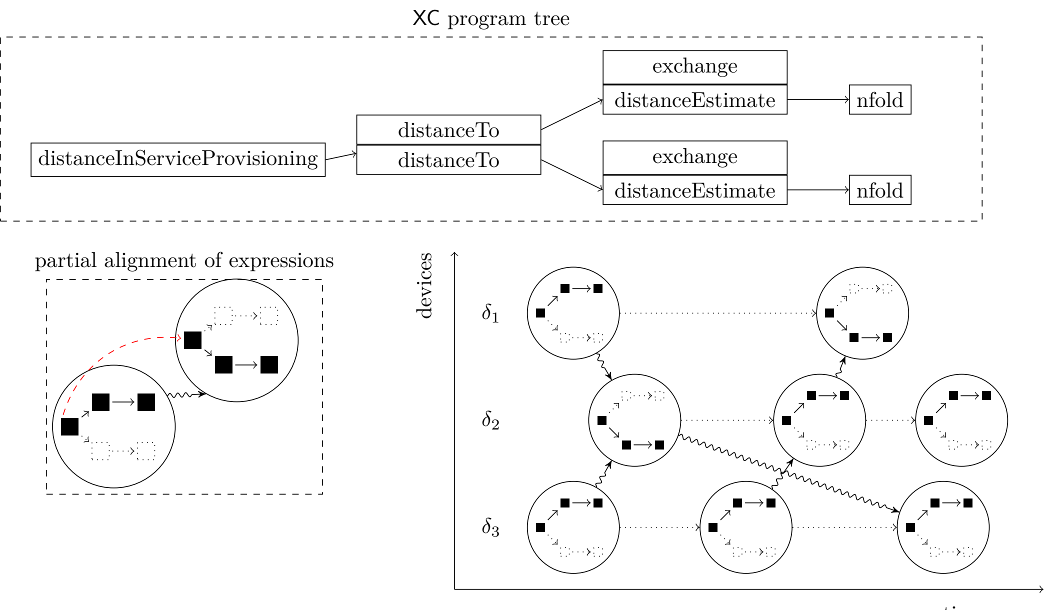


Figure 4 XC alignment mechanism with conditionals for Example 6.

During a round, the program evaluates to `Infinity` on devices where `local` is true. Such devices are considered "obstacles" to avoid. On devices where `local` is false, the program evaluates `distanceTo(gateway)`, which consist of an exchange-expression (c.f. Example 3). Devices in the `local` group do not compute such exchange expression, and do not contribute to the assessment of distances: `distanceTo` is executed in isolation on non-`local`s.

Now suppose we would like the `local` subgroup to compute distances from local `requester` s, and the other subgroup to still compute distances from `gateway`s, excluding in such computations the devices of the complementary group.

```
1  // has type scheme: (bool, bool, bool) → bool
2  def distanceInServiceProvisioning(local, requester, gateway) {
3    if (local) { distanceTo(requester) } else { distanceTo(gateway) }
4  }
```

In this case, in any round, only a single exchange expression is computed, always in the same position in the AST (corresponding to a call of function `distanceTo`). However, the messages exchanged by devices in the `local` group must not be matched with those exchanged by device outside the `local` group, otherwise every device would just compute their distance from the closest local `requester` or non-local `gateway`, which is not the intended behaviour. Luckily, XC grants that this does not happen, as exchange expressions arising from different branches have different stack frames, hence happen in separate interaction domains. ⌟

Figure 4 shows partial alignment for Example 6. At the top, we show the program tree for `distanceInServiceProvisioning`. Note that conditionals are not visible here. Bottom-right, we show a system execution: in each round, only one of the `distanceTo` branches is executed – the branch that has *not* been evaluated is dashed. Bottom-left, we zoom into two rounds of devices that align only partially: they evaluate some common expression which is fully aligned (red dashed arrow), then follow a different branch where there is no alignment. Notice that alignment occurs on the execution of function `distanceInServiceProvisioning` but no actual data is exchanged (since no evaluated `exchange` or `nfold` expression is aligned).

2.6 Fault tolerance in XC

The abstractions discussed so far allow and encourage developers to write XC programs that are resilient to failures. In case a node fails or a message gets lost in inter-node communication, the `exchange` handles the failure transparently from programmers: the node simply does not show up among the neighbours of a given node in the next alignment. With `exchange`, developers specify the logic to *collectively* operate over the neighbours' messages, and make no assumptions on their number or identity, while being encouraged to express the behaviour homogeneously through point-wise operations and `nfold`. As a result, in XC it is idiomatic to write programs with implicit fault tolerance and resilience to devices that dynamically join and leave the set of neighbours (e.g., because they physically change location), transparently from programmers. Programming resilient behaviour can also take advantage of functional composition: simpler resilient blocks can be composed together, building complex applications while retaining fault-tolerance. However, it is important to note that XC does not provide guarantees on fault tolerance by itself. Being a Turing-complete language, non-resilient behaviour can inevitably be programmed, although mostly non-idiomatically: guarantees on idiomatic subsets of the language may be provided, as briefly discussed in Section 4.

3 XC at Work

We now show XC in action by means of example applications in areas like WSNs, IoT, and large-scale CPS. The examples are chosen to *(i)* highlight how composition in XC, enabled by alignment, allows programmers to divide and incrementally deal with the complexity of expressing distributed adaptive behaviour; and *(ii)* show that the expressiveness of XC enables the encoding of advanced algorithms (e.g., with self-organisation properties); and *(iv)* present reusable components used later in our evaluation (Section 6).

▶ **Example 7** (Gossip). The function `gossipEver` spreads the information associated to an event (e.g., pressing a button) through a network. It consists of a single exchange expression executed on every device.

```
1 def gossipEver(event) { // has type scheme: (bool) → bool
2   exchange( False, (n) => retsend nfold(or, n, self(n) or event) )
3 }
```

The first argument of the `exchange` (Line 2) sets the initial value to `False` for `n` (and thus for newly-connected devices, including for the current device in its first round). The second argument is a lambda, whose parameter `n` is the nvalue representing the gossips of neighbouring devices (including the current device itself, for which `n` includes the gossip value in its previous round). Function `nfold` collapses the neighbours' gossips through binary operation `or` (checking whether there is *any* gossip equal to true), with the starting value `self(n) or event` which is true if either the current device had a true gossip in its previous round (i.e., `self(n)` is true) or a true value is fed right now (`event`). The resulting value, the new gossip for the device, is both returned by the function and sent to each neighbour. ◀

Notice that the `gossip` function is agnostic to the network structure and it avoids explicit message management. Its repeated application by a (possibly mobile) network of devices realises the expected behaviour, returning true in every device after a button has been pressed anywhere in the network *as soon as possible*, that is, as soon as the fastest chain of messages from the originating event is able to reach the device.

This function is fully decentralised and every device executes the same logic. Yet, `gossip` only spreads a Boolean event, and once the gossip becomes true, there is no way to flip it to false again. Arbitrary data types and reversibility, require one to break symmetry: some devices (*leaders*) act as sources of truth, and the others will receive their most recent data through a broadcast routine, such as the following.

▶ **Example 8** (Broadcast)**.** Function `broadcast` below implements the propagation of the `value` at nodes of minimal `dist` outwards, along minimal paths ascending `dist`. We assume that `dist` is produced by a function such as `distanceTo` (Example 3).

```
1  def broadcast(dist, value, null) { // has type scheme: (num, A, A) → A
2    val selfRank = (dist, uid);
3    val nbrRank = nbr(selfRank, selfRank);
4    val bestRank = nfold(min, nbrRank, selfRank);
5    val parent = nbrRank == bestRank;
6    exchange( value, (n) =>
7      val selfKey = (value==null, selfRank);
8      val nbrKey = (n==null, nbrRank);
9      val res = snd(nfold( min, (nbrKey, n), (selfKey, value) ));
10     return res
11     send mux(nbr(False, parent), res, null)
12   )
13 }
```

First each device identifies a single *parent device*, as the neighbour having the minimal *rank*, computed in `bestRank` (Line 4). Such rank is a pair of `dist` and `uid` (Line 2), ordered lexicographically, ensuring that the parent is the neighbour of minimal distance to the knowledge source (using `uid` to break ties). The chosen parent is encoded as the only neighbour for which a true value is present in nvalue `parent` (Line 5).

Then, an `exchange` expression sorts out the broadcast received from parent devices, propagating the result to children. The value of the device for the current round is computed in `res` (Line 9), and is taken from the neighbour with the minimum *key*, i.e., minimum rank for a non-`null` value (we assume that `False < True`). For the current device, we use the argument `value` (Line 7). For neighbours, we use the value received from them in `n` (Line 8). The resulting value `res` is returned by exchange and by the whole function, (Line 10). This (possibly band-consuming) value is sent *only* to neighbours which selected the current device as parent, that is, neighbours where `nbr(False, parent)` is true. Every other neighbour receives `null` instead (possibly lighter to transmit): the selection over values and nulls is performed *per-neighbour* by built-in operator `mux` (Line 11). ⌟

The function `broadcast` above uses differentiated messages to neighbours to reduce the network load. This result is achieved by sending values only to the neighbours that actually need them, using placeholder `null` values for the others. In case the message propagation does not need to reach every device of the network, but only some targets, this load can be further reduced by restricting the broadcast into a *channel*, as we explain next.

▶ **Example 9** (Broadcast into a Channel)**.** The function `channelBroadcast` selects a region `channel` of a given `width` connecting a `source` device with a destination device `dest`, and performing a broadcast within the region.

```
1  // has type scheme: (bool, bool, num, A, A) → A
2  def channelBroadcast(source, dest, width, value, null) {
3    val ds = distanceTo(source);
4    val dd = distanceTo(dest);
5    val channel = ds + dd <= broadcast(ds, dd, Infinity) + width;
6    if (channel) { broadcast(ds, value, null) } else { null }
7  }
```

The channel region is computed through the geometrical definition of ellipse (Line 5): the sum of the distances `ds` towards source and `dd` towards destination (computed by `distanceTo`, Lines 3-4) should surpass the distance between source and destination by at most `width` for devices in the channel. The distance between source and destination is obtained through `broadcast(ds, dd, Infinity)`: the parameter `ds` of the broadcast defines that values should be propagated from the `source` outwards; and the value propagated is the parameter `dd`, as it is evaluated in the source (and thus the distance between source and destination). Then, a conditional is used to selectively broadcast the `value` in the source outwards only in the channel region – `null` elsewhere (Line 6).

The example illustrates functional composition: `channelBroadcast` composes several instances of `distanceTo` (Example 3) and `broadcast` (Example 8) to realise a more complex behaviour. Also, the composition preserves the self-organising properties of its constituent parts, hence it able to automatically adapt to changes in `source`, `dest`, `width`, and topology (because, e.g., of mobility or failure).

So far, we have presented functions to build a communication structure to disseminate information over the network. Yet, we haven't addressed the problem of *collecting* such information, especially in the non-trivial case where it is obtained by inspecting the whole network (or part of it).

▶ **Example 10** (Information collection)**.** The `collect` algorithm (inspired by [8]) aggregates the `value` *currently* present in the network, via an arithmetic or an idempotent accumulator, progressively in a network towards a source node – identified as the zero-value of a gradient `dist` (cf. Example 3). The result is updated when values change, unlike Example 7 where a `true` cannot revert to `false`.

```
1  def weight(dist, radius) { // has type scheme: (num,num)→num
2    max(dist-nbr(0,dist),0)*(radius-senseDist)
3  }
4  def normalise(w) { // has type scheme: (num) → num
5    w / nfold(+, w, 0)
6  }
7  // has type scheme: (num, num, A, (A, A) → A, (A, num) → A) → A
8  def collect(dist, radius, value, accumulate, extract) {
9    exchange( value, (n) =>
10     val loc = accumulate(n, value); // local estimate
11     return loc
12     send extract(loc, normalize(weight(dist, radius)))
13   )
14 }
```

The `exchange` construct (Line 9) handles neighbour-to-neighbour propagation of partial accumulates. First, it applies `accumulate` (Line 10) to aggregate the local `value` with the received partial accumulates `n` into `loc`; this is the result of `collect` (Line 11). In other words, the idea is that the local partial accumulate is obtained by accumulating the partial accumulates of neighbours. Then, it computes a normalised weight (Line 12), via functions `weight` and `normalise`, measuring neighbour reliability, using this weight to `extract` from `loc` the partial accumulates to send to neighbours (Line 12). Function `weight` (Line 1) is parametrised by a gradient value `dist` and value `radius` representing the maximum communication range for neighbour interaction; so, the expression is non-negative and the computed weight is larger for neighbours farther from the communication boundaries (i.e., less likely to be lost as neighbours) and closer to the source of the collection. In `normalise` (Line 4), normalisation of weights `w` is achieved by dividing the computed weights

for neighbours by the sum of the neighbours' weights. Depending on the nature of the aggregation (*arithmetic* or *idempotent*, e.g., sum or minimum), different `accumulate` and `extract` functions are used: in the former case, the value is multiplied by the weight:

```
1 def accumulate(v, l) { nfold(+, v, l) } // has type scheme: (A, A) → A
2 def extract(v, w) { v * w }              // has type scheme: (A, num) → A
```

In the latter case, we choose to either send the value or not (also increasing efficiency as in Example 8) depending on whether the weight exceeds a given threshold:

```
1 def accumulate(v, l) { nfold(min, v, l) } // has type scheme: (A, A) → A
2 def extract(v, w) { mux(w >= 0.25, v, Infinity) } // has type scheme: (A, num) → A
```

Improvements over [8] are both stylistic (cleaner code) and in the precision of weights, since in [8] they had to be indirectly (and approximately) deduced on the receiving end. ⌟

▶ **Example 11** (Smart City Monitoring)**.** We consider SmartC, a scenario of smart city monitoring, where devices cooperate with neighbours to process and relay information in the distributed system. This is achieved by the collective execution of an XC program. The system consists of *detectors*, non-mobile nodes (e.g., smart traffic lights) that collect in a bounded surrounding area the contributions of other possibly mobile devices that we call *data-providers* (e.g., buses or people with wearables). Data-providers exhibit a local *warning value*, which signals a need for intervention. Detectors collect warning values and compute a *mean warning* in their area: when the mean warning exceeds some threshold, then they also collect logs from data-providers and dispatch collected data towards the closest *operations centre*. The operations centre might be several hops away from the source, so we want to "broadcast" data hop-by-hop along a short "path" of devices – but without flooding the whole network. The system *(i)* collects and routes data from nodes closer than a certain range towards the closest detector; *(ii)* lets detectors compute the mean levels of warning of the corresponding areas; *(iii)* lets detectors collect and aggregate logs if their mean warning exceeds a certain threshold; and *(iv)* creates self-healing broadcast channels from detectors to the closest operations centres. This logic is implemented by function `smartC` (Figure 5), which reuses `distanceTo`, `collect`, `broadcast` and `channelBroadcast` (Examples 3 and 8–10).

Function `smartC` is defined in terms of local values representing parameters for the algorithm (e.g., `warningThreshold`) or varying inputs (e.g., `localLog`, which denotes a set of log items for a node), which can be thought of as provided by sensors and may change dynamically. The algorithm works as follows. First, a gradient of distances from detectors is computed in the system (Line 3). The nodes that are `inspected` are only those for which the gradient value is less than `inspectionRadius` (Line 4). Then, two different behaviours are defined based on whether a node is inspected or not (Line 6). Nodes not inspected just return `nullReport` (Lines 5 and 18). In the domain of inspected nodes, including the detector, a collection process is activated (Line 7 to 10) in order to let the detector obtain the sum of warning and the number of devices in the area. With such information, the detector can process the mean warning (Line 11) and decide whether the warning level is high (Line 12): such a decision (warning significance) is broadcast from the detector to the rest of the area (Line 13), as a kind of notification to the devices in the surroundings. Also, depending on whether the warning level is high (Line 14 to 16), it either collects the logs from all the nodes in the area (Line 15), or not. In any case, a broadcast on a channel is performed to resiliently communicate the report (set of logs) from the detector to the operations centre (Line 19). ⌟

```
1  def smartC(isDetector, isOpsCentre, channelWidth, inspectionRadius, commRadius,
2             localWarning, warningThreshold, localLog, nullLog, logCat) {
3    val detectorDist = distanceTo(isDetector);
4    val inspected = detectorDist < inspectionRadius;
5    val nullReport = (uid, 0, nullLog);
6    val report = if (inspected) {
7      val (sumWarning,numNodes) = collect(detectorDist, commRadius, (localWarning, 1.0),
8        (v, l) => (nfold(+, fst(v), fst(l)), nfold(+, snd(v), snd(l))),
9        (v, w) => (fst(v)*w, snd(v)*w)
10     );
11     val meanWarning = sumWarning / numNodes;
12     val localWarning = meanWarning > warningThreshold;
13     val warning = broadcast(detectorDist, localWarning, False);
14     val logs = if (warning) {
15       collect(detectorDist, commRadius, localLog, logCat, (v, w) => v)
16     } else { nullLog };
17     (uid, meanWarning, logs)
18   } else { nullReport };
19   channelBroadcast(isDetector, isOpsCentre, channelWidth, report, nullReport)
20 }
```

Figure 5 Possible XC implementation of a smart city monitoring application.

4 Formalisation of XC

In this section we present a formalisation of the core concepts introduced in this paper through Featheweight XC (FXC), a minimal calculus for XC. By virtue of its minimality, FXC is particularly convenient for proving properties both of the language as a whole and of algorithms and fragments of it, such as: type soundness and determinism with respect to let-polymorphic typing, denotational characterisation of expressions as space-time values [7], with functional compositionality of global behaviour. We further discuss XC expressivity and resilience properties (inherited from results in literature) in Section 7.

4.1 Syntax

Figure 6 (top) shows the syntax of FXC. As in [34], the overbar notation indicates a (possibly empty) sequence of elements, e.g., $\overline{x}$ is short for $x_1, \dots, x_n$ $(n \geq 0)$. Note that the syntax induces a standard functional language, with no peculiar features for distribution: distribution is nonetheless apparent in the operational semantics. An FXC *expression* $\mathtt{e}$ can be either:

- a *variable* $\mathtt{x}$;
- a (possibly recursive) *function* $\mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$, which may have free variables;
- a *function call* $\mathtt{e}(\overline{\mathtt{e}})$;
- a *let-style* expression $\mathtt{val}\ \mathtt{x} = \mathtt{e}; \mathtt{e}$;
- a *local literal* ℓ, that is either a built-in function $\mathtt{b}$, a defined function $\mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$ *without* free variables, or a data constructor $\mathtt{c}$ applied to local literals (possibly none);
- an *nvalue* $\mathtt{w}$, as described in Section 2.2.

FXC can be typed using standard let-polymorphism for higher-order languages, without distinguishing between types for local values and types for neighbouring values. This is accomplished by promoting local values to nvalues, and designing constructs and built-in functions of the language to always accept nvalues for their arguments (more details on this in Section 4.2, Device semantics). As local and neighbouring types are not distinguished by FXC, in this section we avoid underlying neighbouring values and their types. Free variables are defined in a standard way (Figure 6, middle), and an expression $\mathtt{e}$ is *closed* if $\mathsf{FV}(\mathtt{e}) = \emptyset$. Programs are closed expressions without nvalues as sub-expressions. Indeed, nvalues only arise in computations, and are the only values produced by evaluating (closed) expressions.

Syntax:

$\mathtt{e} ::= \mathtt{x} \mid \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\} \mid \mathtt{e}(\overline{\mathtt{e}}) \mid \mathtt{val}\ \ \mathtt{x} = \mathtt{e};\mathtt{e} \mid \ell \mid \mathtt{w}$		expression
$\mathtt{w} ::= \ell[\overline{\delta} \mapsto \overline{\ell}]$		nvalue
$\ell ::= \mathtt{b} \mid \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\} \mid \mathtt{c}(\overline{\ell})$		local literal
$\mathtt{b} ::= \mathtt{exchange} \mid \mathtt{nfold} \mid \mathtt{self} \mid \mathtt{updateSelf} \mid \mathtt{uid} \mid \ldots$		built-in function

Free variables of an expression:

$\mathsf{FV}(\mathtt{x}) = \{\mathtt{x}\} \quad \mathsf{FV}(\ell) = \mathsf{FV}(\mathtt{w}) = \emptyset \quad \mathsf{FV}(\mathtt{fun}\ \mathtt{x}_0(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{\mathtt{e}\}) = \mathsf{FV}(\mathtt{e}) \setminus \{\mathtt{x}_0, \ldots, \mathtt{x}_n\}$

$\mathsf{FV}(\mathtt{e}_0(\mathtt{e}_1, _, \mathtt{e}_n)) = \bigcup_{i=0_n} \mathsf{FV}(\mathtt{e}_i) \qquad \mathsf{FV}(\mathtt{val}\ \ \mathtt{x} = \mathtt{e};\mathtt{e}') = \mathsf{FV}(\mathtt{e}) \cup \mathsf{FV}(\mathtt{e}') \setminus \{\mathtt{x}\}$

Syntactic sugar:

$(\overline{\mathtt{x}})\ \texttt{=>}\ \mathtt{e} \quad ::= \mathtt{fun}\ \mathtt{y}(\overline{\mathtt{x}})\{\mathtt{e}\}$ *where* $\mathtt{y}$ *is a fresh variable*

$\mathtt{def}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\} \quad ::= \mathtt{val}\ \mathtt{x} = \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\};$

$\mathtt{if}(\mathtt{e})\{\mathtt{e}_\top\}\ \mathtt{else}\ \{\mathtt{e}_\bot\} ::= \mathtt{mux}(\mathtt{e}, ()\ \texttt{=>}\ \mathtt{e}_\top, ()\ \texttt{=>}\ \mathtt{e}_\bot)()$

Figure 6 Syntax (top), free variables (middle) and syntactic sugar (bottom) for FXC expressions.

The syntax in Figure 6 (top) diverges partially from the one used in Sections 2 and 3. However, the full syntax of XC can be recovered by defining missing constructs as syntactic sugar. Besides some standard simplifications (infix notation for binary operators, omitted parenthesis in 0-ary constructors, implicit `pair` constructor), some non-trivial encoding is described in Figure 6 (bottom). In particular, lambda expressions can be converted into fun-expressions with a fresh name, and defined functions can be encoded as a let expression binding the function name. Branching can be encoded by abstracting the code in the branches, selecting one of them with the `mux` operator and then applying it.

4.2 Operational semantics

The operational semantics is defined as *(i)* a big-step *device semantics*, providing a formal account of the computation of a device within one round; and *(ii)* a small-step *network semantics*, formalising how different device rounds communicate.

Device semantics. Figure 7 presents the device semantics, formalised by judgement $\delta;\sigma;\Theta \vdash \mathtt{e} \Downarrow \mathtt{w};\theta$, to be read as "expression $\mathtt{e}$ evaluates to nvalue $\mathtt{w}$ and value-tree θ on device δ with respect to sensor values σ and value-tree environment Θ", where:

- $\mathtt{w}$ is called the *result* of $\mathtt{e}$;
- θ is an ordered tree with nvalues on some nodes (cf. Figure 7 (top)), representing messages to be sent to neighbours by tracking the nvalues produced by exchange-expressions in $\mathtt{e}$, and the stack frames of function calls;
- Θ collects the (non expired) value-trees received by the most recent firings of neighbours of δ, as a map $\delta_1 \mapsto \theta_1$, ..., $\delta_n \mapsto \theta_n$ $(n \geq 0)$ from device identifiers to value-trees.

We assume every function expression $\mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$ occurring in the program is annotated with a unique name τ before the evaluation starts. Then, τ will be the name for the annotated function expression $\mathtt{fun}^\tau \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$, and $\mathtt{b}$ the name for a built-in function $\mathtt{b}$.

The syntax of value-trees and value-tree environments is in Fig. 7 (top). The rules for judgement $\delta;\sigma;\Theta \vdash \mathtt{e} \Downarrow \mathtt{v};\theta$ (Fig. 7, middle) are standard for functional languages, extended to evaluate a sub-expression $\mathtt{e}'$ of $\mathtt{e}$ w.r.t. the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of

Auxiliary definitions:

$\theta ::= \langle\overline{\theta}\rangle \mid \mathtt{w}\langle\overline{\theta}\rangle$ value-tree $\qquad \sigma$ sensor state

$\Theta ::= \overline{\delta} \mapsto \overline{\theta}$ value-tree environment $\qquad \delta$ device identifier

$$\pi_i(\langle\theta_1, \dots, \theta_n\rangle) = \theta_i \qquad \pi_i(\mathtt{w}\langle\theta_1, \dots, \theta_n\rangle) = \theta_i \qquad \pi_i(\overline{\delta} \mapsto \overline{\theta}) = \overline{\delta} \mapsto \pi_i(\overline{\theta})$$

$$\overline{\delta} \mapsto \overline{\theta}\,|_{\mathtt{f}} = \left\{\delta_i \mapsto \theta_i \;\mid\; \theta_i = \mathtt{w}\langle\overline{\theta}'\rangle,\ name(\mathtt{w}(\delta_i)) = name(\mathtt{f})\right\} \qquad \begin{array}{l} name(\mathtt{b}) = \mathtt{b} \\ name(\mathtt{fun}^\tau\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}) = \tau \end{array}$$

Evaluation rules: $\boxed{\delta;\sigma;\Theta \vdash \mathtt{e} \Downarrow \mathtt{w};\theta}$

$$\text{[E-NBR]}\ \frac{}{\delta;\sigma;\Theta \vdash \mathtt{w} \Downarrow \mathtt{w};\langle\rangle} \qquad \text{[E-LIT]}\ \frac{}{\delta;\sigma;\Theta \vdash \ell \Downarrow \ell[];\langle\rangle}$$

$$\text{[E-VAL]}\ \frac{\delta;\sigma;\pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \mathtt{w}_1;\theta_1 \qquad \delta;\sigma;\pi_2(\Theta) \vdash \mathtt{e}_2[\mathtt{x} := \mathtt{w}_1] \Downarrow \mathtt{w}_2;\theta_2}{\delta;\sigma;\Theta \vdash \mathtt{val}\ \mathtt{x} = \mathtt{e}_1;\mathtt{e}_2 \Downarrow \mathtt{w}_2;\langle\theta_1,\theta_2\rangle}$$

$$\text{[E-APP]}\ \frac{\delta;\sigma;\pi_{i+1}(\Theta) \vdash \mathtt{e}_i \Downarrow \mathtt{w}_i;\theta_i \quad \text{for all } i \in 0,\dots,n \qquad \delta;\sigma;\pi_{n+2}(\Theta\,|_{\mathtt{f}}) \vdash \mathtt{f}(\mathtt{w}_1,\dots,\mathtt{w}_n) \Downarrow^* \mathtt{w}_{n+1};\theta_{n+1} \text{ where } \mathtt{f} = \mathtt{w}_0(\delta)}{\delta;\sigma;\Theta \vdash \mathtt{e}_0(\mathtt{e}_1,\dots,\mathtt{e}_n) \Downarrow \mathtt{w}_{n+1};\mathtt{f}[]\langle\theta_0,\dots,\theta_{n+1}\rangle}$$

Auxiliary evaluation rules: $\boxed{\delta;\sigma;\Theta \vdash \mathtt{f}(\overline{\mathtt{w}}) \Downarrow^* \mathtt{w};\theta}$

$$\text{[A-FUN]}\ \frac{\delta;\sigma;\Theta \vdash \mathtt{e}[\mathtt{x} := \mathtt{fun}^\tau\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}, \overline{\mathtt{x}} := \overline{\mathtt{w}}] \Downarrow \mathtt{w};\theta}{\delta;\sigma;\Theta \vdash \mathtt{fun}^\tau\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}(\overline{\mathtt{w}}) \Downarrow^* \mathtt{w};\theta} \qquad \text{[A-UID]}\ \frac{}{\delta;\sigma;\Theta \vdash \mathtt{uid}() \Downarrow^* \delta;\langle\rangle}$$

$$\text{[A-XC]}\ \frac{\Theta = \overline{\delta} \mapsto \overline{\mathtt{w}}\langle\dots\rangle \quad \mathtt{w} = \mathtt{w}_i[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)] \qquad \delta;\sigma;\pi_1(\Theta) \vdash \mathtt{w}_f(\mathtt{w}) \Downarrow (\mathtt{w}_r,\mathtt{w}_s);\theta}{\delta;\sigma;\Theta \vdash \mathtt{exchange}(\mathtt{w}_i,\mathtt{w}_f) \Downarrow^* \mathtt{w}_r;\mathtt{w}_s\langle\theta\rangle} \qquad \text{[A-SELF]}\ \frac{}{\delta;\sigma;\Theta \vdash \mathtt{self}(\mathtt{w}) \Downarrow^* \mathtt{w}(\delta);\langle\rangle}$$

$$\text{[A-FOLD]}\ \frac{\Theta = \delta_1 \mapsto \theta_1,\dots,\delta_n \mapsto \theta_n \quad \ell_0 = \mathtt{w}_3(\delta) \qquad \delta;\sigma;\emptyset \vdash \mathtt{w}_1(\ell_{i-1},\mathtt{w}_2(\delta_i)) \Downarrow \ell_i[];\theta \text{ if } \delta_i \neq \delta \text{ else } \ell_i = \ell_{i-1}}{\delta;\sigma;\Theta \vdash \mathtt{nfold}(\mathtt{w}_1,\mathtt{w}_2,\mathtt{w}_3) \Downarrow^* \ell_n[];\langle\rangle} \qquad \cdots$$

Figure 7 Device (big-step) operational semantics of FXC.

Θ. This *alignment* process is modelled by the auxiliary "projection" functions π_i (for any positive natural number i) (Fig. 7, top). When applied to a value-tree θ, π_i returns the i-th sub-tree θ_i of θ. When applied to a value-tree environment Θ, π_i acts pointwise on the value-trees in Θ.

The alignment process ensures that the value-trees in the environment Θ always correspond to the evaluation of the same sub-expression currently being evaluated. To ensure this match holds (as said before, of the stack frame and position in the AST), in the evaluation of a function application $\mathtt{f}(\overline{\mathtt{w}})$, the environment Θ is reduced to the smaller set $\Theta\,|_{\mathtt{f}}$ of trees which corresponded to the evaluation of a function with the same *name* (as defined in Fig. 7 (top)).

Rule [E-NBR] evaluates an nvalue $\mathtt{w}$ to $\mathtt{w}$ itself and the empty value-tree. Rule [E-LIT] evaluates a local literal ℓ to the nvalue $\ell[]$ and the empty value-tree. Rule [E-VAL] evaluates a val-expression, by evaluating the first sub-expression with respect to the first sub-tree of the environment obtaining a result $\mathtt{w}_1$, and then the second sub-expression with respect to the second sub-tree of the environment, after substituting the variable $\mathtt{x}$ with $\mathtt{w}_1$.

Rule [E-APP] is standard eager function application: the function expression $\mathtt{e}_0$ and each argument $\mathtt{e}_i$ are evaluated w.r.t. $\pi_{i+1}(\Theta)$ producing result $\mathtt{v}_i$ and value-tree θ_i. Then, the function application itself is demanded to the auxiliary evaluation rules, w.r.t. the last sub-tree of the trees corresponding to the same function: $\pi_{n+2}(\Theta\,|_{\mathtt{f}})$. The auxiliary rule [A-FUN] handles the application of fun-expression, which evaluates the body after replacing the

Network configuration (sensors/environment/result fields) and action labels:

$$\Sigma \;::=\; \overline{\delta} \mapsto \overline{\sigma} \quad \text{sensors field} \qquad N \;::=\; \langle \Sigma; \Psi; \psi \rangle \quad \text{network configuration}$$
$$\Psi \;::=\; \overline{\delta} \mapsto \overline{\Theta} \quad \text{environment field}$$
$$\psi \;::=\; \overline{\delta} \mapsto \overline{\mathtt{w}} \quad \text{result field} \qquad act \;::=\; \delta \;\big|\; \delta\delta' \;\big|\; conf \quad \text{action label}$$

Notations for restriction and update of a sensors/environment/result field m:

$$m|_X = m' \quad \text{s.t.} \quad \mathbf{dom}(m') = \mathbf{dom}(m) \cap X \text{ and } m'(\delta) = m(\delta)$$
$$m[m'] = m'' \text{ s.t. } \mathbf{dom}(m'') = \mathbf{dom}(m) \cup \mathbf{dom}(m'), \;\; m''(\delta) = \begin{cases} m'(\delta) \text{if } \delta \in \mathbf{dom}(m') \\ m(\delta) \text{ otherwise} \end{cases}$$

Transition rules: $\boxed{N \xrightarrow{act} N}$

$$\text{[N-FIRE]} \; \frac{\Theta = \text{filter}(\Psi(\delta)) \quad \delta; \Sigma(\delta); \Theta \vdash \mathtt{e_{main}} \Downarrow \mathtt{w}; \theta \quad \Theta' = \Theta[\delta \mapsto \theta]}{\langle \Sigma; \Psi; \psi \rangle \xrightarrow{\delta} \langle \Sigma; \Psi[\delta \mapsto \Theta']; \psi[\delta \mapsto \mathtt{w}] \rangle}$$

$$\text{[N-RECV]} \; \frac{\theta = \Psi(\delta)(\delta) \quad \Theta' = \Psi(\delta')[\delta \mapsto \theta]}{\langle \Sigma; \Psi; \psi \rangle \xrightarrow{\delta\delta'} \langle \Sigma; \Psi[\delta' \mapsto \Theta']; \psi \rangle} \qquad \text{[N-CONF]} \; \frac{\overline{\delta} = \mathbf{dom}(\Sigma') \quad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle \Sigma; \Psi; \psi \rangle \xrightarrow{conf} \langle \Sigma'; \Psi_0[\Psi|_{\overline{\delta}}]; \psi|_{\overline{\delta}} \rangle}$$

Figure 8 Network (small-step) operational semantics of FXC.

arguments $\overline{\mathtt{x}}$ with their provided values $\overline{\mathtt{w}}$, and the function name $\mathtt{x}$ with the fun-expression itself. Rules [A-UID] and [A-SELF] trivially encode the behaviour of the `uid` and `self` built-ins. Rule [A-XC] evaluates an exchange-expression, realising the behaviour described at the beginning of Section 2.3. Notation $\mathtt{w}_1[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)]$ is used to represent the nvalue $\mathtt{w}_1$ after the update for each i of the message for δ_i with the custom message $\mathtt{w}_i(\delta)$. The result is fed as argument to function $\mathtt{w}_f$: the first element of the resulting pair is the overall result, while the second is used to tag the root of the value-tree. Rule [A-FOLD] encodes the `nfold` operators. First, the domain of Θ is inspected, giving a (sorted) list $\delta_1, \dots, \delta_n$. An initial local value ℓ_0 is set to the "self-message" of the third argument. Then, a sequence of ℓ_i is defined, each by applying function $\mathtt{w}_1$ to the previous element in the sequence and the value $\mathtt{w}_2(\delta_i)$ (skipping δ itself). The final result ℓ_n is the result of the application, with empty value-tree. Auxiliary rules for the other available built-in functions are standard, do not depend on the environment, hence have been omitted.

Network semantics. The evolution of a whole network of devices executing a program $\mathtt{e_{main}}$ is formalised by transitions $N \xrightarrow{act} N'$, which reads *"network configuration N evolves to network configuration N' by a transition with label act"*. The syntax of network configurations and action labels is in Figure 8 (top). A network configuration N is a triple $\langle \Sigma; \Psi; \psi \rangle$, where:

- Σ maps each device δ of the network to a sensors status σ, representing the status of sensors of δ at a given time (for any choice of a representation of sensor status σ);
- Ψ maps each device δ of the network to a value-tree environment Θ, collecting the (non expired) value-trees received by the most recent firings of neighbours of δ;
- ψ is a partial mapping that, at any given time, maps devices δ of the network to the nvalue $\mathtt{w}$ produced by their most recent firings (if any such firing already happened).

We remark that, for each device δ, the sensors status $\Sigma(\delta)$, the value-tree environment $\Psi(\delta)$ and the nvalue $\psi(\delta)$ are locally stored in the device δ – there is no global memory.

Each transition $N \xrightarrow{act} N'$ consists of one of these three different evolution steps:

- if $act = \delta$, it formalises the round of device δ, and the memorisation of the resulting nvalue $\mathtt{w}$ and value-tree θ in the device's local store;

- if $act = \delta\delta'$ with $\delta \neq \delta'$, it formalises that device δ' receives a value-tree θ from δ;
- if $act = conf$, it formalises an overall change of the network configuration as (possible) change of sensor status of devices and (possible) entering/leaving of devices in the network.

A sequence of transitions $\langle\emptyset;\emptyset;\emptyset\rangle \xrightarrow{act_1} \dots \xrightarrow{act_n} N_n$ thus represents the operational evolution of a network. The transition rules of the semantics of a program $\mathtt{e}_{\mathtt{main}}$ are given in Figure 8 (bottom). Rule [N-FIRE] formalises a computation round of device δ: given the locally-available sensors status $\Sigma(\delta)$ and value-tree environment filtered out of expired value-trees $\Theta = \text{filter}(\Psi(\delta))$, it uses the device semantics judgement to obtain the nvalue $\mathtt{w}$ and value-tree θ produced by the round. Then, it uses $\mathtt{w}$ to update $\psi(\delta)$, and uses θ to update $\Theta(\delta)$ (thus modelling immediate reception of the self-message). The filtering function $\text{filter}(\cdot)$ is a parameter of the calculus, meant to clear out old stored values from the value-tree environments in Ψ, usually based on space/time tags attached to value-trees.

Rule [N-RECV] formalises the reception of a value-tree from device δ by another device δ'. The message conceptually dispatched is the value-tree θ corresponding to δ obtained from the value-tree environment $\Psi(\delta)$ of δ itself. On the recipient side, the received message θ is locally associated to δ in the value-tree environment $\Psi(\delta')$ of δ'. Even though rule [N-RECV] dispatches the same message θ to any recipient δ', an optimised implementation could compress received messages by collapsing each received nvalue $\mathtt{w}$ within θ to the message $\mathtt{w}(\delta')$ for δ', discarding the rest before storing it in the local memory.

Rule [N-CONF] formalises an update of the sensor status of devices and entering/leaving of devices (auxiliary notations $m\,|_X$ and $m[m']$ are in Fig. 8, second frame, representing domain restriction and pointwise update of maps). Given a new sensors mapping Σ', the resulting network configuration contains exactly the devices $\overline{\delta}$ in the domain $\mathbf{dom}(\Sigma')$ of Σ'. This is achieved by reducing the result field ψ to the new set of devices through $\psi\,|_{\overline{\delta}}$, constructing an environment field Ψ_0 mapping every $\overline{\delta}$ to the empty environment $\emptyset$, then reducing the existing environment field Ψ to the new set of devices through $\Psi\,|_{\overline{\delta}}$, and finally using this to overwrite the values in Ψ_0. Note that the reboot of a device δ can be modelled by two applications of rule [N-CONF]: one removing δ from the network configuration and another re-inserting it. When a device δ is removed from the network, the content of its local memory (sensors, messages, result) are lost.

5 Implementation

We implemented a Scala and a C++ version of XC. The Scala version has been developed as an extension of ScaFi [22], and aims at showcasing the DSL and maximize portability to different platforms, including simulators. The C++ version has been developed as an extension of FCPP [5], and has consequently been integrated into the main FCPP distribution. This version targets performance and devices with limited resources. Running experiments on real IoT devices with the C++ version is still work in progress.

5.1 Scala DSL

We provide an implementation of XC as a DSL embedded into the Scala language[3] because of its cross-platform support [30], popularity for building distributed systems [33], and advanced support for internal DSLs [4]. This implementation has been developed as an extension of

[3] The Scala DSL is publicly available under the Apache 2.0 license at `https://github.com/scafi/artifact-2021-ecoop-xc` and permanently as an archived artifact on Zenodo [21].

ScaFi [22]. The DSL is organized into a few core XC constructs and a library of reusable functions. The core constructs (cf. Figure 6) are declared by a Scala trait with the following interface:

```
1 trait XCLang {
2   def branch[T](cond: NValue[Boolean])(th: => NValue[T])(el: => NValue[T]): NValue[T]
3   def exchange[T](init: NValue[T])(f: NValue[T] => (NValue[T],NValue[T])): NValue[T]
4 }
```

The `if`/`else` of XC is modelled as a `branch` function to avoid conflicts with Scala's `if`. The two branches are call-by-name parameters, as usual. A neighbour value is implemented as a class with a default message and a concrete map of messages for other devices.

```
1 class NValue[T](val defaultMessage: T, val customMessages: Map[ID,T] = Map.empty) {
2   def fold[V>:T](init: V)(f: (V,V)=>V): NValue[V] = // ...
3   def map2[R,S](other: NValue[R])(m: (T,R)=>S): NValue[S] = // ...
4   // more built-ins ... (cf. Figure 2)
5 }
```

We leverage Scala implicit conversions and extension methods [25], imported by mixing in `XCLib`, to automatically convert values of type `T` to `NValue`s of `T`s and, e.g., to extend `NValue`s of `Numeric`s to accept operators like `+` (to combine nvalues point-wise). An abstract class `XCProgram[T]` requires programmers to override the method `main:T`. Moreover, it exposes methods `sense` and `senseNeighbour` to subclasses for retrieving local and neighbouring values from the execution environment. For instance, the gradient program (Example 3) can be encoded as follows.

```
1 object gradient extends XCProgram[Double] with XCLib {
2   def main =
3     exchange(Double.PositiveInfinity)(n =>
4       mux(sense[Boolean]("source")){ 0.0 }{
5         (n + senseNeighbour("distance")).fold(Double.PositiveInfinity)(Math.min)
6       })
7 }
```

An `XCProgram[T]` models a single local computation. As discussed (Section 2.1), a XC system involves multiple devices repeatedly acquiring context, computing the round, and propagating messages to neighbours. The execution environment provides a context with values from the sensors for the built-in sensing functions (cf. Figure 2) and with the messages from the neighbours. For example, the following code shows the execution on a device:

```
1 while(true) {
2   val sensorData = getData() // implementation-specific
3   val messagesFromNeighbours = getMessages() // implementation-specific
4   val context = Context(sensorData, messagesFromNeighbours)
5   val (output, messageCollection) = gradient.fire(context)
6   process(output) // implementation-specific
7   propagate(messageCollection) // implementation-specific
8 }
```

Note that in this implementation message communication occurs only *before* (Line 3) and *after* (Line 7) the firing (Line 5) to ensure that the `exchange` within the round are all executed atomically w.r.t. the messages that are received and sent by the device (Section 2.1). The details of a system implementation depends on the target deployment. Example deployments that could be implemented include a peer-to-peer network of IoT devices (where each node handles computation and communication with neighbours), a collection of thin IoT devices connected to the cloud (where only sensor and actuator data flows between the IoT nodes and the cloud, which is responsible for running computations and internally handling the

```
1 FUN bool gossipEver(ARGS, bool event){ CODE
2   return nbr(CALL, false, [&](field<bool> n){
3     return any_hood(CALL, n) or event;
4   });
5 }
```

Figure 9 Implementation of the `gossipEver` function in C++/XC.

message passing), or a simulator (where physical and/or logical devices are virtualised). What these implementations must do in order to support a XC system is providing the implementation-specific functions of the listing above: `getData()` to obtain values from the local environment, `getMessages()` to retrieve messages from neighbours (e.g., a peer node may keep them in a buffer, a cloud platform may use an in-memory database service, a simulator may use an ad-hoc map-like data structure), `process()` to drive actuations (e.g., locally on a node, or through a command on a cloud back-end), and `propagate()` to send exported data to neighbours (e.g., through a direct message to the neighbour, or through a write on shared state in simulations or cloud).

5.2 C++ DSL

We implemented XC as a C++ DSL[4], by extending FCPP [5]. This implementation is designed for *(i)* efficiency, and *(ii)* custom architectures. For *(i)*, we rely on C++'s compile-time optimization and execution on the bare metal. We also performed careful profiling to manually optimize crucial parts of the library. For neighbouring values, we use `vector<T>` (having two sorted lists for ids and values) from C++ STL – which is more efficient than hash maps for linear folding and point-wise operations. For communication, we serialise messages and pass them to the network driver (for low level devices this is usually a non-standard API where one can configure the byte content of the message and the transmission power). For *(ii)* we exploit that C/C++ compilers are usually available for custom architectures, while also aiming to minimise the amount of dependencies, to ease the deployment. For instance, the implementation includes its own serialisation header, compile-time type inspection utilities, multi-type valued maps, option types, quaternions, tagged tuples, etc.

Compared to the Scala implementation, the embedding of XC into C++ is more verbose, thus requiring additional effort for development (see Figure 9 for a code sample). We are currently working on testing this implementation on several different back-ends, including:

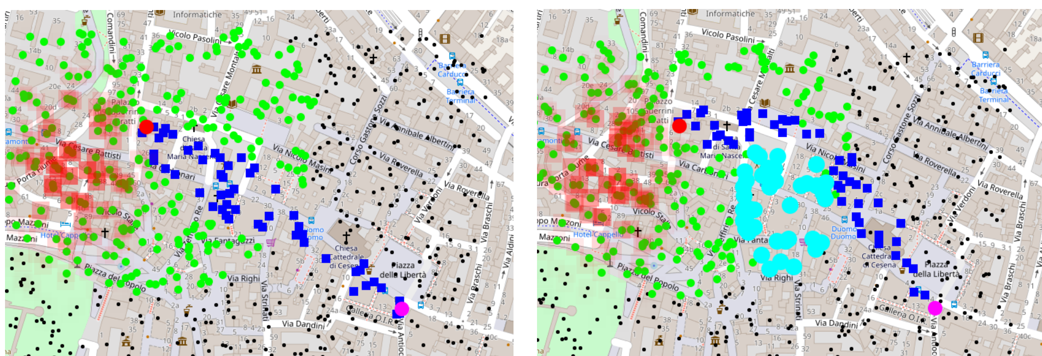
- processing of XC algorithms on large graph-based data in HPC;
- deployment on microcontroller architectures with either Contiki OS or MIOSIX.

No external dependencies are needed for those back-ends.

6 Evaluation

In this section, we evaluate XC.[5] The goal is to show that *(RQ1)* the decentralised execution of the XC program on each device results in the desired *collective* behaviour and that *(RQ2)* the overall behaviour can be expressed by composing functions of collective behaviour that

[4] The C++ DSL is publicly available under the Apache 2.0 license at: `https://fcpp.github.io`.
[5] The simulation framework, its description, and instructions for reproducing the experiments are publicly available at `https://github.com/scafi/artifact-2021-ecoop-smartc` and permanently as an archived artifact on Zenodo [20].



(a) Communication structures are in place (inspection area and channel from detector to operations centre). Sensors detect some dangerous situation.

(b) A blackout destroys the original channel. The channel self-repairs by circumventing the obstacle.

Figure 10 Two snapshots of the SmartC case study.

correctly combine thanks to alignment. The evaluation does not focus on the efficiency of fault recovering because this aspect is application-dependent – not language-dependent. For instance, the recovery time for a channel depends on the algorithms used to compute distances and broadcasts, relative to the network assumptions.

SmartC case study. We consider a simulation of the SmartC scenario described in Example 11, and we implement it both in the Scala and C++ DSLs (the results in this section refer to the Scala implementation). We believe that other application domains, such as cyber-physical systems (CPS) and wireless sensor networks (WSN), would not pose fundamentally different challenges compared to the considered scenario: WSN focus on information flows, which is part of the case study, and CPS emphasize on actuation, which could be a simple variant of the scenario, e.g., where agents move according to the gathered reports. In the simulation setup, 600 devices each running the XC program communicate with every neighbour currently in a 50-metre range once per second. We consider a single detector and a single operations centre. The simulator enables the collection of data exported at the individual nodes (i.e., the program Example 11 is extended with simulation-specific code). We measure, every second, the actual (instantaneous) mean warning in the inspection area (using an oracle, namely a process that can inspect the simulated system at any instant) and the mean warning measured by the operations centre. We consider the average result over 30 simulations varying the actual displacement of devices and scheduling offsets. We inject a blackout event that disconnects a set of devices from the system, hence disrupting the channel. Figure 10 shows two snapshots of the simulation with devices (black dots), detector (red dot), sensors within the area inspected by the detector (green dots), operations centre (magenta dot), and inoperable devices (cyan dots). Semi-transparent red squares denote the warning level locally perceived by sensors. Blue squares are nodes in the channel from detector to operations centre.

Results. Figure 11 shows that the mean warning received by the operations centre (blue) during a run approximates the actual warning in the inspected area (magenta). The jags in the second blue wave are due to perturbations (exacerbated by the obstacle) that temporarily destroy the channel in a few simulations, while delays depend on the firing frequency and communication hops (from inspection area edges to detector to operations centre).

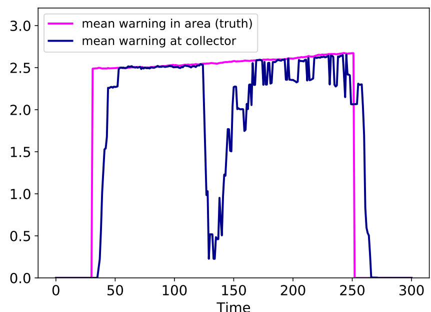


Figure 11 Execution of SmartC.

For *(RQ1)*, this result shows that the XC program enables the system to self-organise in such a way that the operations centre can acquire the mean warning level aggregated by the detector, in spite of environmental changes and perturbations induced by mobility and failure. For *(RQ2)*, we remark that the self-organising behaviour resulting from the XC program in SmartC is achieved by direct composition of several reusable blocks of collective behaviour, namely `distanceTo`, `broadcast`, `collect`, and `channelBroadcast` (cf. Example 11).

Comparison with other programming models. Additionally, to get a sense of the benefit of the XC implementation w.r.t. other programming models, we re-implemented functions `distanceTo` and `channel` (a version of `channelBroadcast` without the final broadcast) with actors and pub-sub[6]. Essentially, the intuition of the XC advantage in terms of expressiveness lies in the implicit declaration of data exchange for each building block usage, instead of the more explicit and verbose message handling/sending (for actors) and topic forging with event consuming/producing (for pub-sub). Despite field calculi have been studied in a series of papers [49], no systematic comparison (e.g. via formal translation) with other approaches has been previously carried out. There are two challenges: *(i)* very few works target the kind of distributed systems (e.g. self-organisation) targeted by XC, hence a comparison needs to consider general-purpose languages and a wide spectrum of software designs; *(ii)* system behaviour unfolds by the interplay of device semantics *and* network semantics (cf. Section 4.2), which are brittle to neatly separate in other approaches. Though, we can here focus on a comparison among *(i)* a pub-sub "idiomatic" solution, S_{PS}, *(ii)* a pub-sub XC-like solution S_{PSXC} with a design inspired by XC, and *(iii)* an XC solution S_{XC}. This allows us to draw some interesting indications on the compactness that programming in XC can provide.

The core programs are 82, 28, 22 LoC long. In S_{PS}, the logic spreads over multiple subscription handlers, while in S_{PSXC} and S_{XC} the core logic is neatly separated. The S_{PS} version uses 4 handlers (and crucially, any additional field would need a further handler), 2 `send`s, and 4 `publish`es, while S_{PSXC} uses 2, 1, and 3 resp. Also, S_{PS} keeps 6 state variables for the input context of a device–S_{PSXC} only 3. W.r.t. S_{XC}, S_{PSXC} has a coding overhead due to the topics management and to the more brittle handling of neighbour data, of about 27% more LoC, 73% more words, and 35% more method calls. Finally, the main limitation of S_{PS} is the loss of compositionality, the inter-dependence between the different computations of fields, and the fragility that stems from the management of change propagation.

[6] The paradigm comparison is publicly available at:
`https://github.com/metaphori/aggregate-paradigm-comparison`.

7 Related work

We organize related work by first providing a high-level perspective on field-based coordination. Next we describe approaches based on ensembles and attribute-based communication, which are close to our solution but adopt fundamentally different design choices. Finally, we compare in detail with field calculi and briefly discuss abstraction and compositionality.

Field-based coordination. Field-based coordination, as a paradigm to develop self-organising systems, originate from two main research areas: *spatial computing* [29], where the idea of *aggregate computing* [16] emerged, and *coordination models and languages* [39]. Two surveys cover these two perspectives. The work in [15] reviews various DSLs ranging from multi-agent modelling to WSNs with respect to how they measure and manipulate space-time, model physical evolution and computation, and (meta-)manipulate computation itself. More recently, [49] outlines the historical development from tuple-based and field-based coordination to field calculi, covering the state of the art and future challenges within aggregate computing research. The latter work also reviews various formalisations of field computations. As discussed later, XC subsumes the constructs of field calculi as of [48, 6] and so has a potential as foundation for field-based coordination, and as *lingua franca* to describe distributed algorithms for large-scale systems, and specifically for self-organisation.

Ensembles and attribute-based communication. Recently, field-based coordination is also framed as a paradigm for collective adaptive systems (CAS) [32], which is a further application target for self-organisation techniques in general. There, related approaches include *ensemble-based engineering* [19, 27] and *attribute-based communication* [1]. Ensemble approaches leverage the notion of *ensemble*, i.e., a dynamic group of components typically specified through a membership relationship, for CAS programming. De Nicola et al. propose SCEL [27], a process-algebraic approach where systems are made of components, i.e., processes with an attribute-based interface for addressing their state (knowledge) and evolving by executing actions on predicated groups of target components; actions provide ways to read, retrieve, put information, and to create new components. AbC [1] captures the essence of attribute-based interaction of SCEL: components are (parallel compositions of) processes associated with an attribute environment, and actions are guarded through predicates over such attributes. Attribute-based communication approaches exploit attributes labelling devices and matching mechanisms to dynamically define sets of recipients for multi-casts, to promote coordination in CASs. This is also possible in field calculi, but it is made much simpler by the selective communication mechanism in XC, a key contribution of this paper.

Field calculi. Field calculi, surveyed in [49], assume a neighbouring relationship for connectivity and, upon that, enable defining dynamic groups of devices by exploiting branching and recursion. However, interaction is not based on attribute matching but on execution of the same functions (alignment) involving communication constructs like `exchange`.

In the following we compare XC with the *field calculus (FC)* [48, 6], which is the reference model for computational fields [49], also implemented by DSLs like ScaFi [22, 23] and FCPP [5, 13]. FC features two separate kinds of values (and types): *local values* (of *local* type) and *neighbouring values* (of *field* type). XC combines these into a single class of nvalues $\mathtt{v} = \ell[\overline{\delta} \mapsto \overline{\ell}]$. In particular, *local values* are equivalent to nvalues $\ell[]$ without custom messages, and *neighbouring values* are equivalent to nvalues with any valid default message. This unification allows a simpler type system and, crucially, differentiated messages to neighbours.

By interpreting FC values as nvalues, all FC message-exchanging constructs (`nbr`, `rep` [48] and `share` [6]) can be modelled within XC: `nbr` is the same defined function introduced in Section 2.2, just restricted to operate on local values only; `share` corresponds to an `exchange` with `retsend` restricted to operate on local values only; and $\texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \Rightarrow \texttt{e}_2\}$ can be translated to $\texttt{exchange}(\texttt{e}_1, (\texttt{x}) \Rightarrow \texttt{retsend}\ \texttt{e}_2[\texttt{x} := \texttt{self}(\texttt{x})])$. Notice that the converse translation is not possible, as `nbr`, `rep` or `share` expressions with arguments of neighbouring type have no defined behaviour in FC. Thus, `nbr` and `exchange` in XC are strictly more expressive than their corresponding FC counterparts `nbr` and `share`: they can be used with expressions producing nvalues with custom messages to model differentiated messages.

The properties for subsets of the *field calculus* (FC), as surveyed in [49], include eventual recovery and stabilisation after transient changes (self-stabilisation) [48], independence of the results from the density of devices [17], real-time error guarantees [12], efficient monitorability of spatio-temporal logic properties [9, 11], and ability to express all physically consistent computations (space-time universality) [7]. The fact that every FC program can be encoded within XC, automatically imports all these results into XC and paves the way towards future extensions to XC programs not expressible in FC.

Abstraction and compositionality. XC's mechanism to send and receive messages to/from neighbours provides a high-level programming model for message passing which abstracts over failures (cf. Section 2.6) and is reminiscent of shared memory models. Namely: *(i)* nodes work on a fixed snapshot of incoming messages once the round starts (because message exchange occurs only between rounds) and *(ii)* messages can be overwritten or read multiple times until they expire, resulting in a model similar to shared memory. This combination, thanks to the alignment property (a distinctive feature of XC and field calculi, which enables functional composition as illustrated in Sections 2.4 and 2.5), achieves an abstraction level that it is not available in the competing spatial computing approaches (surveyed, e.g., in [15, 49]) or shared memory models (surveyed, e.g., in [42, 43]).

8 Conclusion and Outlook

In this paper, we introduce the design of XC, a programming language for homogeneous distributed systems that abstracts over a number of traditional issues in developing distributed applications, including faults, lost messages, and asynchronicity. XC's minimal design features only one communication primitive. We show that despite its simplicity, XC can capture a number of communication patterns in homogeneous distributed systems and it is effective for writing large scale distributed software.

The design of XC, through nvalues and the new semantic construct `exchange`, opens interesting directions for future work. First, we plan to characterise XC programs enjoying two fundamental properties: self-stabilisation [48], and density independence [17], as the ability of a field computation to converge with the density of devices filling space. Second, works such as [48] define combinators, namely, general field functions implementing key behavioural elements of information diffusion, collection, and degradation, the composition of which turns out to define a number of interesting higher-level functions. We plan to devise new such building blocks with XC, e.g. to realise *sparse choice* of leaders [41] and consensus [14]. Finally, we are currently assessing the impact of XC constructs on real-world application programming, thanks to our porting in Scala and C++.

References

1. Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. Programming interactions in collective adaptive systems by relying on attribute-based communication. *Science of Computer Programming*, 192, 2020. doi:10.1016/j.scico.2020.102428.
2. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Comput.*, 9(1):37–49, 1995. doi:10.1007/BF01784241.
3. Joe Armstrong. Erlang. *Commun. ACM*, 53(9), September 2010. doi:10.1145/1810891.1810910.
4. Cyrille Artho, Klaus Havelund, Rahul Kumar, and Yoriyuki Yamagata. Domain-specific languages with Scala. In *ICFEM*, volume 9407 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2015. doi:10.1007/978-3-319-25423-4_1.
5. Giorgio Audrito. FCPP: an efficient and extensible field calculus framework. In *Proceedings of the 1st International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS*, pages 153–159. IEEE Computer Society, 2020. doi:10.1109/ACSOS49614.2020.00037.
6. Giorgio Audrito, Jacob Beal, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Field-based coordination with the share operator. *Logical Methods in Computer Science*, 16(4), 2020. doi:10.23638/LMCS-16(4:1)2020.
7. Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. Space-time universality of field calculus. In *Coordination Models and Languages*, volume 10852 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2018. doi:10.1007/978-3-319-92408-3_1.
8. Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Optimal resilient distributed data collection in mobile edge environments. *Computers & Electrical Engineering*, 2021. doi:10.1016/j.compeleceng.2021.107580.
9. Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Volker Stolz, and Mirko Viroli. Adaptive distributed monitors of spatial properties for cyber-physical systems. *J. Syst. Softw.*, 175:110908, 2021. doi:10.1016/j.jss.2021.110908.
10. Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. Compositional blocks for optimal self-healing gradients. In *Self-Adaptive and Self-Organizing Systems (SASO), 2017*, pages 91–100. IEEE, IEEE Computer Society, 2017. doi:10.1109/SASO.2017.18.
11. Giorgio Audrito, Ferruccio Damiani, Volker Stolz, Gianluca Torta, and Mirko Viroli. Distributed runtime verification by past-CTL and the field calculus. *J. Syst. Softw.*, 187:111251, 2022. doi:10.1016/j.jss.2022.111251.
12. Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Enrico Bini. Distributed real-time shortest-paths computations with the field calculus. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 23–34. IEEE Computer Society, 2018. doi:10.1109/RTSS.2018.00013.
13. Giorgio Audrito, Luigi Rapetta, and Gianluca Torta. Extensible 3D simulation of aggregated systems with FCPP. In *24th International Conference on Coordination Models and Languages, Proceedings*, Lecture Notes in Computer Science. Springer, 2022. To appear.
14. Jacob Beal. Trading accuracy for speed in approximate consensus. *Knowledge Eng. Review*, 31(4):325–342, 2016. doi:10.1017/S0269888916000175.
15. Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. doi:10.4018/978-1-4666-2092-6.ch016.
16. Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the Internet of Things. *IEEE Computer*, 48(9), 2015. doi:10.1109/MC.2015.261.
17. Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution in the Internet of Things. *ACM Transactions on Autonomous and Adaptive Systems*, 12(3):12:1–12:29, 2017. doi:10.1145/3105758.

18 Arne Brutschy, Giovanni Pini, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. Self-organized task allocation to sequentially interdependent tasks in swarm robotics. *Auton. Agents Multi Agent Syst.*, 28(1):101–125, 2014. doi:10.1007/s10458-012-9212-y.

19 Tomás Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: an ensemble-based component system. In *Symposium on Component Based Software Engineering (CBSE)*, pages 81–90. ACM, 2013. doi:10.1145/2465449.2465462.

20 Roberto Casadei. scafi/artifact-2021-ecoop-smartc: v1.2, 2022. doi:10.5281/ZENODO.6538822.

21 Roberto Casadei. scafi/artifact-2021-ecoop-xc: v1.2, 2022. doi:10.5281/ZENODO.6538810.

22 Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. FScaFi : A core calculus for collective adaptive systems programming. In *ISoLA (2)*, volume 12477 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 2020. doi:10.1007/978-3-030-61470-6_21.

23 Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Engineering collective intelligence at the edge with aggregate processes. *Eng. Appl. Artif. Intell.*, 97:104081, 2021. doi:10.1016/j.engappai.2020.104081.

24 Shane S. Clark, Jacob Beal, and Partha P. Pal. Distributed recovery for enterprise services. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, September 21-25, 2015*, pages 111–120. IEEE Computer Society, 2015. doi:10.1109/SASO.2015.19.

25 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 341–360. ACM, 2010. doi:10.1145/1869459.1869489.

26 Soura Dasgupta and Jacob Beal. A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pages 7282–7287. IEEE, 2016. doi:10.1109/CDC.2016.7799393.

27 Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Trans. Auton. Adapt. Syst.*, 9(2):7:1–7:29, 2014. doi:10.1145/2619998.

28 Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys*, 47(4), May 2015. doi:10.1145/2716320.

29 André DeHon, Jean-Louis Giavitto, and Frédéric Gruau, editors. *Computing Media and Languages for Space-Oriented Computation*, volume 06361 of *Dagstuhl Seminar Proceedings*, 2007. URL: http://drops.dagstuhl.de/portals/06361.

30 Sébastien Doeraene. Cross-platform language design in Scala.js (keynote). In *SCALA@ICFP*, page 1. ACM, 2018. doi:10.1145/3241653.3266230.

31 Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), June 2003. doi:10.1145/857076.857078.

32 Alois Ferscha. Collective adaptive systems. In *UbiComp/ISWC Adjunct*, pages 893–895. ACM, 2015. doi:10.1145/2800835.2809508.

33 Debasish Ghosh, Justin Sheehy, Kresten Krab Thorup, and Steve Vinoski. Programming language impact on the development of distributed systems. *J. Internet Serv. Appl.*, 3(1):23–30, 2012. doi:10.1007/s13174-011-0042-y.

34 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001. doi:10.1145/503502.503505.

35 Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005. doi:10.1145/1082469.1082470.

36 Pushpendu Kar, Arijit Roy, and Sudip Misra. Connectivity reestablishment in self-organizing sensor networks with dumb nodes. *ACM Trans. Auton. Adapt. Syst.*, 10(4):28:1–28:30, 2016. doi:10.1145/2816820.

37 Naomi Kuze, Daichi Kominami, Kenji Kashima, Tomoaki Hashimoto, and Masayuki Murata. Hierarchical optimal control method for controlling large-scale self-organizing networks. *ACM Trans. Auton. Adapt. Syst.*, 12(4):22:1–22:23, 2018. doi:10.1145/3124644.

38 Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. Asynchronous distributed execution of fixpoint-based computational fields. *Log. Methods Comput. Sci.*, 13(1), 2017. doi:10.23638/LMCS-13(1:13)2017.

39 Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994. doi:10.1145/174666.174668.

40 Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Pervasive Computing and Communications, 2004*, pages 263–273. IEEE, 2004. doi:10.1109/PERCOM.2004.1276864.

41 Yuanqiu Mo, Jacob Beal, and Soura Dasgupta. An aggregate computing approach to self-stabilizing leader election. In *International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 112–117. IEEE, 2018. doi:10.1109/FAS-W.2018.00034.

42 Christine Morin and Isabelle Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Trans. Parallel Distributed Syst.*, 8(9):959–969, 1997. doi:10.1109/71.615441.

43 Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *International Design and Test Workshop (IDT)*, pages 12–17. IEEE, 2011. doi:10.1109/IDT.2011.6123094.

44 Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Workshop on Data Management for Sensor Networks*, pages 78–87, 2004. doi:10.1145/1052199.1052213.

45 Torsten Hoefler on behalf of the MPI Forum. MPI: A message-passing interface standard, version 2.2. Specification, Message Passing Interface Forum, September 2009. URL: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

46 H. Van Dyke Parunak, Sven Brueckner, Robert S. Matthews, and John A. Sauter. Pheromone learning for self-organizing agents. *IEEE Trans. Syst. Man Cybern. Part A*, 35(3):316–326, 2005. doi:10.1109/TSMCA.2005.846408.

47 Rajiv Ranjan, Omer F. Rana, Surya Nepal, Mazin Yousif, Philip James, Zhenya Wen, Stuart L. Barr, Paul Watson, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, Massimo Villari, Maria Fazio, Saurabh Kumar Garg, Rajkumar Buyya, Lizhe Wang, Albert Y. Zomaya, and Schahram Dustdar. The next grand challenges: Integrating the internet of things and data science. *IEEE Cloud Comput.*, 5(3):12–26, 2018. doi:10.1109/MCC.2018.032591612.

48 Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2):16:1–16:28, 2018. doi:10.1145/3177774.

49 Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.*, 109, 2019. doi:10.1016/j.jlamp.2019.100486.

50 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, 2018. doi:10.1145/3276499.

PEDroid: Automatically Extracting Patches from Android App Updates

Hehao Li ✉
Shanghai Jiao Tong University, China

Yizhuo Wang ✉
Shanghai Jiao Tong University, China

Yiwei Zhang ✉
Shanghai Jiao Tong University, China

Juanru Li ✉ 🏠
Shanghai Jiao Tong University, China

Dawu Gu ✉
Shanghai Jiao Tong University, China

Abstract

Identifying and analyzing code patches is a common practice to not only understand existing bugs but also help find and fix similar bugs in new projects. Most patch analysis techniques aim at open-source projects, in which the differentials of source code are easily identified, and some extra information such as code commit logs could be leveraged to help find and locate patches. The task, however, becomes challenging when source code as well as development logs are lacking. A typical scenario is to discover patches in an updated Android app, which requires bytecode-level analysis. In this paper, we propose an approach to automatically identify and extract patches from updated Android apps by comparing the updated versions and their predecessors. Given two Android apps (original and updated versions), our approach first identifies identical and modified methods by similarity comparison through code features and app structures. Then, it compares these modified methods with their original implementations in the original app, and detects whether a patch is applied to the modified method by analyzing the difference in internal semantics. We implemented PEDroid, a prototype patch extraction tool against Android apps, and evaluated it with a set of popular open-source apps and a set of real-world apps from different Android vendors. PEDroid identifies 28 of the 36 known patches in the former, and successfully analyzes 568 real-world app updates in the latter, among which 94.37% of updates could be completed within 20 minutes.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases Diffing, Patch Identification, Android App Analysis, App Evolution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.21

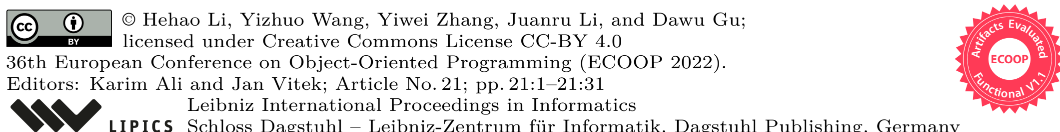
Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.24`

Funding This work was supported by the National Key Research and Development Program of China (No.2020AAA0107803).

Acknowledgements We are grateful to our reviewers for their valuable support and suggestions.

1 Introduction

Android apps nowadays are published at an unprecedented rate and many developers frequently update their apps for a variety of reasons such as helping maintain the robustness or introducing more competitive features. An update usually leads to multiple modifications of the app, some of which are used to improve the functionality or performance, while a significant type of modifications is to fix bugs in apps. This type of modifications, also known as *patches*, reflect how the developers fix the bug. Researchers not only learn the causes



© Hehao Li, Yizhuo Wang, Yiwei Zhang, Juanru Li, and Dawu Gu;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 21; pp. 21:1–21:31
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of bugs but also discover and fix similar bugs [19, 23, 22] in other apps through analyzing the information carried by patches. However, it is often unclear for analysts how Android app developers repair existing defects for lack of detailed commit logs, especially for security participants who do not have access to the source code. Thus, the gap between the updated apps and patches hinders the analysis of patches.

To the best of our knowledge, few approaches effectively identify patches against Android *updates* (i.e., the original and updated versions of an app). A common and simple way to retrieve existing patches is crawling from bug-tracking systems of open-source projects, such as GitHub Issue Tracker [16], where the detailed commit messages or bug reports are available to determine whether the modified methods contain patches. This approach does not work on closed-source apps that have less information to explain the reasons for updates. The descriptions about the updates of closed-source apps often only claim what feature has been added or some bugs have been repaired, but do not further explain the type, cause, and repair information of the bugs. On the other hand, compared with the open-source project, the closed-source app has a much larger amount and accounts for the majority of Android apps. As for binary-level analysis, SPAIN [45] focuses on patches in C binaries, but the huge difference between procedure-oriented and object-oriented program languages makes it unable to apply on Android apps.

Another problem to identify patches at bytecode level is how to locate modified methods in updates. Previous works [45, 38] of patch analysis on C binary utilize BinDiff [7] to achieve the goal. However, there exist few accurate diffing tools on bytecode of Android apps, due to the popularity of code obfuscation (e.g., using ProGuard [30] to protect bytecode). Most works only implement coarse-grained similarity comparison [6, 49, 39, 47] cross apps, which cannot locate the modified methods between two versions of an app, while other works [20, 43, 33] link the original methods with their updated versions by method names which cannot resist obfuscation techniques.

To address the above problems, in this paper, we propose a bytecode-level patch extraction approach, named PEDroid, to automatically locate the patches in updates of Android apps. The workflow of PEDroid consists of two phases: 1) locating the modified methods in two versions of an app, and 2) identifying patches among the modified methods. In phase 1, given the original and updated versions of an Android app, PEDroid first calculates the method-level *matching relations* based on features extracted from bytecode and the structure of the app. The method-level matching relation refers to the two versions of the same method, including identical and modified methods. With the matching relations, it filters out the identical methods whose features are identical and focuses on the modified methods. To identify patches in phase 2, we propose an effective approach to determine the patches from two aspects: 1) the call sites of the modified methods, and 2) the difference in internal semantics. In particular, PEDroid analyzes the call sites of the modified methods using a static taint analysis to check whether the methods use *external values* (i.e., external inputs or results from other methods). Then, it compares the internal semantics of the two versions of the modified methods through aligning the same operations of external values within the two methods and analyzing the modification related to these operations. Finally, PEDroid identifies the patches whose modification is used to fix the processing logic before these operations or handle the errors generated by them.

We evaluated PEDroid on two datasets of Android apps: the first set contains 13 updates of popular open-source apps, and the second one contains 568 real-world updates. We first tested PEDroid on the open-source dataset to evaluate its effectiveness. PEDroid achieves a recall of 92% in differential analysis, and successfully identifies 28 of 36 patches in patch identification. The results show that our approach effectively locates the modified methods and identifies patches. Then, PEDroid ran on the second dataset and successfully

extracted 98,591 patches. Through a further manual analysis, we confirmed several types of patches including security check addition, date usage correcting, error handling, etc. For the time cost, 63.91% of the updates were analyzed within 5 minutes, 83.98% were completed within 10 minutes, and 94.37% were completed within 20 minutes. It shows that PEDroid is capable of discovering rich types of patches in real-world apps.

In summary, our work includes the following contributions:

- We propose a novel approach to extract patches from the neighboring versions of Android apps, and implement PEDroid based on the approach, which labels the identical and modified methods in given APK files, and then identifies patches among all modified methods. To the best of our knowledge, PEDroid is the first work that extracts patches from updates of close-sourced Android apps.
- Due to the lack of a standard benchmark to evaluate the accuracy of differential analysis and patch identification, we collected a dataset with 13 updates of 6 popular open-source apps, which contains 36 patches and 47 non-bugfix updates. The dataset can be used as a benchmark for future works to evaluate the performance of patch extraction.
- We also evaluate the applicability of PEDroid on 568 real-world app updates. 98,591 patches are discovered by PEDroid, including various types (e.g. adding security checks, correcting data usage). All updates are successfully analyzed and 94.37% can be completed within 20 minutes.

2 Related Works

2.1 Diffing in Android

Diffing is a common technique to compare the difference between two programs. There are numerous works to diff two versions of a program at the source code level. Git-diff tool [11] defaults input is sequential and cannot handle the changes in text order, for example, the different order of methods in a class between compilation. Furthermore, it cannot resist the broadly-used renaming obfuscation (e.g., ProGuard[30]) for sensitiveness to all characters in the text. GumTree [9] diffs two versions of abstract syntax tree (AST) of a single Java source code file and considers the different order. However, it provides only a fine-grained diffing between two class files but no method-level matching relations on apps. To retrieve matching relations, some works [32, 33, 43] link two versions of a method by defined patterns, and involves method names in patterns or similarity comparison. But it cannot either handle changes that do not follow these patterns or deal with bytecode with little symbolic information. Schäfer et al. [31] propose an approach to extract matching relations of methods in framework by their usage (e.g. calling and extension) in apps, which builds on the framework or test cases provided by developers. But for all methods in apps, a large proportion will be ignored by the approach. Therefore, these existing diffing tools cannot meet our requirements to locate the modified methods on bytecode.

Apart from these diffing tools, there are many bytecode-level approaches to detect similarity between two Android apps. Many previous works only extract coarse-grained features from code to resist obfuscation. For example, only method signatures are extracted as code features in several works [6, 49, 39, 47], which makes them unable to discover the modification within a method. To achieve the goal of comparing the similarity at the method level, SimiDroid [20] defaults the two methods with the same signatures (i.e., class name, method name, parameter and return types) as matched methods. Hence, the approach cannot resist renaming obfuscation. Another similarity comparison technique [8] only focusing on single methods also obtains inaccurate results. For example, method `a` and `b` of class `A` in the updated version are matched with method `b` of class `B` and method `c` of class `C` in the original version. Therefore, a more precise approach to matching at the method level is necessary.

2.2 Patch Identification

Most existing works on patch analysis focus on open-source projects. The keyword-based approach is the most common way to identify patches, and they collect patches directly from open-source project repositories by parsing reports with predefined keywords (e.g., bug, error and fault) in their issue tracking systems [26, 24, 37, 21, 17, 40]. Different from open source projects that provide formatted and exact code update information, released apps usually do not provide detailed descriptions about changed methods. Instead, they just give some brief comments about update information[1] or even nothing [29]. Hence, it is hard to locate relevant code snippets just by these text descriptions. In addition, Xinda Wang et al. [38] adopt a matching learning-based technique to identify security patches in open-source C projects. They conclude basic, syntactic, and semantic features of changes and train models by open-source patch datasets. However, due to the commercial competition between apps and the prevention of attackers carrying out attacks, few developers open security issues to promote research and analysis. Therefore, the lack of datasets makes it difficult to implement effectively on closed-source Android apps.

As for previous efforts at binary level, Xu et al. [44] generate function signatures for known patches to match, which is unlikely to discover unknown patches. SPAIN [45] identifies patches based on the heuristic that patches are less likely to introduce new semantics than other modifications, and they use the difference of registers, flags, and memory between before and after code snippets to represent the semantics. However, since the object-oriented program language (e.g., Java) is used, most registers in Android apps point to object references, and operations are usually implemented by API or method invocation instead of calculation. Therefore, the semantics of Android bytecode cannot be represented by numerical differences and such an approach is inapplicable in Android apps. To our best knowledge, there is no effective way to identify patches on Android apps.

3 Overview

The goal of our work is to understand patches and the corresponding bugs, and automatically extract patches from Android app updates. While there are a variety of ways to do so, we seek to design an applicable, automated and systematic approach. In this section, we first discuss various challenges we need to solve (Section 3.1), then give corresponding solutions against these challenges (Section 3.2), and finally describe the overview of our tool (Section 3.3).

3.1 Challenges

There will be a number of challenges in order to achieve our goal and these include:

Challenge 1. How to obtain code features. In order to retrieve matching relations, we first calculate code feature similarity. One of the most used code features between two version apps is the sequences of instructions, which describes the project updates by comparing the text line by line [11]. Another common code feature is method signature [20, 43, 33]. However, both the two features could not be applied to represent Android bytecode due to the compilers, obfuscators and even developer customization. Hence, only code order or the method signatures is not feasible in our work. Therefore, we have to first determine how to retrieve the code features.

[1] App developers usually describe the app update briefly (e.g., "Fixed some bugs") in the `WHAT'S NEW` section of a mobile app homepage.

Challenge 2. How to retrieve the matching relations. Having the method features, the next step is to retrieve the matching relations to locate the methods that are of our interest. Since the patches are usually used to update apps, we focus on the modified methods. Unfortunately, existing studies could not retrieve matching relations at the method level concretely. Some works only detect re-used components (e.g., third-party library) by coarse-fine similarity comparison [6, 49, 39, 47] or retrieve specific matched methods by patterns and method name [20, 43, 33]. Hence, a more precise approach to matching at the method level is necessary.

Challenge 3. How to identify patches in modified methods. Having obtained the modified methods, we still need to further identify the patches. Since the lack of commit logs and open-source databases, the existing works [26, 24, 37, 21, 17, 40] cannot be applied to Android updates. And other approaches are also inapplicable because of the huge difference between procedure-oriented language and object-oriented program languages [45] or the aim to discover specific patches against our purpose [44]. Hence, how to identify the patches from modified methods is another challenge.

3.2 Solutions

As previously mentioned, if we intend to perform patch identification in Android apps, we have to face lots of challenges. Fortunately, we have obtained the following insights to address the above challenges.

Solution 1. Extracting features after removing noisy changes. Instead of calculating similarity directly on bytecode through code instruction sequences and method signatures, we combine multiple strategies to extract stable code features which eliminate the noisy changes caused by obfuscation and compilation. Specifically, two steps are involved. First, we replace volatile identifiers with specific labels to resist renaming obfuscation. Second, we divide bytecode into different code units and sort order-independent units, including basic blocks[2], fields and methods, to normalize the order.

Solution 2. Matching guided by positional relationships. We observed that *most of the code is identical between app updates, especially for the updates with small version upgrades.* Thus, to pinpoint the matching relations and further locate the modified methods, our key insight is to utilize the positional relationships in the program structure to assist in matching the modified code. Specifically, we first locate packages containing identical code features in different versions as matched packages. And then we utilize the package hierarchy [3] of the matched packages and similarity comparison to determine the matching relations of other packages. All matched packages are used to further determine the matching relations of classes and methods. Finally, those matched methods with different features are considered as modified methods.

Solution 3. Identifying patches by pinpointing buggy operation. Most unexpected behaviors of the methods are caused by the incorrect handle of the input, and the corresponding patches in the updated version are used to fix incorrect usage or handle the errors. Especially,

[2] a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
[3] a tree of packages and their subpackages. It is like directory structures.

the input comes from not only external inputs (e.g., network I/O and user interaction) but also unexpected results returned from other methods. We call them *external values*. Our insight to identifying the patch is that *a patch usually fixes the processing logic before the buggy operation or handles the errors generated by the buggy operation, while the target of operation tends to involve external values*. Thus, we try to locate the buggy operation to identify patches. To achieve it, we first analyze the usage of the modified methods to check whether they use the external values, then align the original operations of external values within the two methods, and finally determine the patch by specific semantic changes. Such changes are indicated by the original operations which have different dependencies between two versions or result in extra error handling (i.e., exit or exception capture) of the method, and the operation is pinpointed as a buggy operation.

Example. To better illustrate the insight used in Solution 3, we give the motivating examples in Figure 1. The example in Figure 1a fixes the processing logic for the input by adding checks. In this case, the parameter path is the input of the method, and it usually

```
1     private void patch1(String path) {
2         File file = new File(path);
3 +       if(file.exists()) {
4             file.delete();
5 +       }else{
6 +           Log.e("Tag", "Cannot find target 
7 +       }
8     }
```

(a) Fix processing logic before a buggy op

```
1     private void patch2(String path) {
2         File file = new File(path);
3 +       try {
4             file.delete();
5 +       } catch (Exception e){
6 +           Log.e("Tag", "Cannot delete target file.");
7 +       }
8     }
```

(b) Handle errors generated by a buggy operation.

Figure 1 Examples of two types of patches. Statements with green background are added snippets in updated version.

3.3 Framework Overview

Based on the solutions to the three challenges, we design PEDroid, the first patch extraction tool on Android updates. Figure 2 depicts the workflow of PEDroid, which consists of two phases:

1. **Differential analysis.** PEDroid first establishes the structure of apps and extracts features of disassembly code (in Section 4.1). Then, it uses the package as the unit to match between the two versions of the app (in Section 4.2), and finally extracts the matching relations at the method level (in Section 4.3).
2. **Patch identification.** PEDroid extracts the modified methods in the results of differential analysis, and checks whether it is affected by external values at each call site (in Section 5.1). It then locates the operation of the external values within the method and analyzes the modification related to the operations. PEDroid reports the patch if the modification is used to fix the processing logic or handle the errors (in Section 5.2).

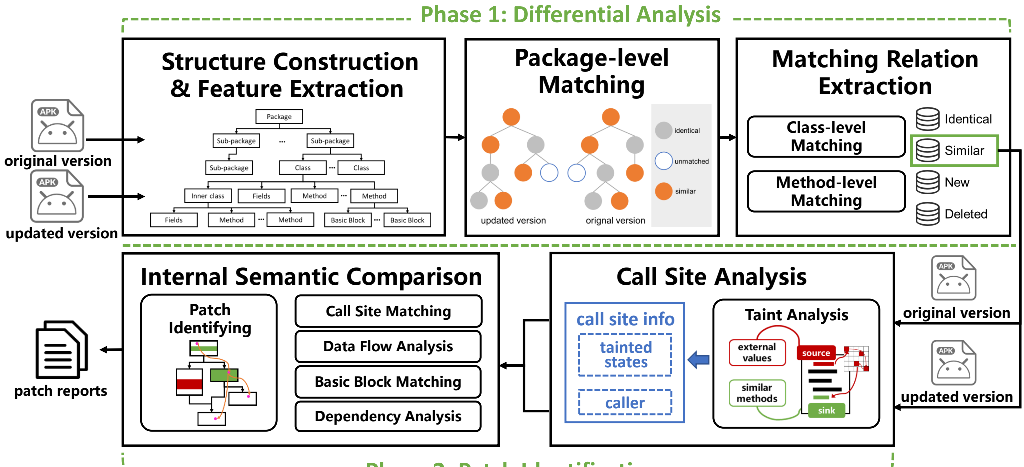


Figure 2 The workflow of PEDROID.

4 Differential Analysis

In this section, we present the design principles of differential analysis, as well as the adopted techniques. PEDROID retrieves method-level matching relations between APK updates through three steps: structure construction and feature extraction, package-level matching, and matching relation extraction.

4.1 Structure Construction & Feature Extraction

The first step of differential analysis is to disassemble the Android app and establish the app structure, including package hierarchy, classes, and code elements in classes (e.g., methods). First, PEDROID builds the relations among packages and classes by the directory structures of the disassembled app, where directories correspond to packages and files correspond to classes. Then, it parses the file content and extracts details of each class, such as fields and methods. Especially, since many nested classes (e.g., inner classes, local classes, anonymous classes, and lambda expressions) contain less information, matching them respectively will lead to false positives. To eliminate it, PEDROID recovers the nested relations and treats them as subunits of the classes they belong to. In detail, PEDROID retrieves it through system annotations from the decompiled class files, i.e., `Ldalvik/annotation/MemberClasses`, `Ldalvik/annotation/EnclosingClass`, `Ldalvik/annotation/EnclosingMethod`.

After app structure construction, PEDROID builds code features from the bottom up according to the structure. Specifically, we adopt two strategies to make the feature stable.

1. **Replacing volatile identifiers.**
   To remove the volatile parts in code, we use the specific labels to fuzz types and the instructions. First, because types contain volatile identifiers, PEDROID only retains the primitive types and framework types, and replaces others by label X to remove the noise

Listing 1 Example for fuzzy type. `Landroid/content/Context` is a framework-type and `V` (i.e., `void`) is a primitive type. `Lcom/text/example` is replaced by `X`.

```
Original:   <init>(Landroid/content/Context;Lcom/test/example;)V
Fuzzy   :   <init>(Landroid/content/Context;X)V
```

Table 1 Rules for fuzzy instruction.

Type	Label	Original instruction	Fuzzy instruction
Register	R	mov v0, v1	mov R, R
Label	L	if-eqz :const_0	if-eqz :L
Resouce ID	N	const v0, 0x7f112222	const R, N
Method/Class (except Android API)	X	invoke-virtual p0, Lcom/test/example;->call()V	invoke-virtual R, X

brought by the identifiers, when extracting types involving some code elements such as fields. In this way, PEDroid converts them into the *fuzzy type*. For example, List 1 gives an example of fuzz types in a method signature. For instructions, PEDroid replaces the different types of the operand with the different labels, as shown in Table 1. Each processed instruction is called *fuzzy instruction*.

In detail, PEDroid extracts the following feature elements for different code units:

- **Basic Block.** The feature of a basic block consists of all the fuzzy instructions in the basic block.
- **Method.** The feature of a method includes method access flags, fuzzy types of all parameters, and the features of all basic blocks in the method.
- **Field.** The feature of a field is a string consisting of access flags, fuzzy type, and the non-default initialization value. The default initialization values (i.e. null, ‘’, 0, etc.) and names of fields are ignored.
- **Class.** The feature of a class includes the fuzzy types of superclass and interfaces, the features of fields, methods, and nested classes.

2. **Normalizing orders.**
The order-independent features such as the features of basic blocks and methods are sorted to normalize the order. It is because the extracted features without normalizing will be different because of the different orders between the two versions. Since these changes are caused by compilation rather than developers, we eliminate them. To normalize the order of fuzzy instructions with a basic block, PEDroid analyzes the dependencies of registers and sorts the order of sequential instructions without dependencies on each other. For independent units (including basic blocks, methods, fields, and classes), PEDroid directly sorts the features of the same types of the included units. For example, the features of basic blocks are sorted and then become a part of the method feature.

After extracting features and normalizing the order, PEDroid calculates the overall feature of each unit by hashing all the orderly features to represent the unit. Hence, the overall feature of a unit is calculated based on the overall hash of the included units, rather than all the feature elements of each included unit. And PEDroid records the overall features and feature elements of all units and the inclusion relations between the units.

4.2 Package-level Matching

With the app structure and the features of code elements, PEDroid calculates the matching relations between packages based on the package hierarchy, which is the sub-graph of the app structure. Specifically, PEDroid extracts identical classes, which are the two classes with identical features. And then it locates *identical packages* having at least one identical class. Among the rest packages, PEDroid utilizes their positional relations with the identical packages on the two package hierarchy to search for matching candidates, and treats the packages with the greatest similarity as *similar packages*. In summary, it includes two steps: *identical package matching* and *similar package matching*.

Identical Package Matching

PEDroid builds an identical package collection PKG_{iden}, which stores the identical package pairs. To achieve it, PEDroid first finds out the identical classes. Especially, only when the overall feature of the class in the updated version is unique and the same as the unique feature in the original version, the two version classes are regarded as identical classes. Packages with one or multiple identical classes are considered identical, and the two packages are added to PKG_{iden} as a pair. According to these rules, PEDroid obtains the matching pair collection PKG_{iden} of the identical packages, which maps an updated package to all the original packages considered to be identical. That means a package may have multiple identical classes to different packages of another version.

Similar Package Matching

Based on the identical package collection PKG_{iden} and package hierarchy, PEDroid matches similar packages by different positional relationships. Algorithm 1 represents our approach to determine similar packages from candidates. In detail, PEDroid first discovers the candidates by the positions of matched packages (which are initially identical packages) on package hierarchy and then selects the packages with the greatest similarity among candidates as similar packages.

Algorithm 1 Searching similar packages in all candidates.

```
Input: Candidates set Candidate_sim
Output: Similar packages PKG_simi
PKG_simi ← ∅
map_1 : mapping new version packages to all candidates packages in old version
map_2 : mapping old version packages to all candidates packages in new version
for ⟨p_1, p_2⟩ in Candidate_sim do
    map_1[p1].add(p2)
    map_2[p2].add(p1)
end
for ⟨p_1, candidates_1⟩ in map_1 do
    p_2 ← get most similar package in candidates_1 of p_1
    candidates_2 ← map_2[p_2]
    p_1' ← get most similar package in candidates_2 of p_2
    if p_1 == p_1' then
        PKG_simi.add(⟨p_1, p_2⟩)
    end
end
return PKG_simi
```

Similarity Calculation. PEDroid quantifies similarity based on the similarity between features. Since the feature is extracted from the bottom up, the similarity between the upper units involves their bottom units. That means, before calculating the similarity of the units, the matching relations between their included units should be obtained. For example, the similarity of classes is calculated based on the matching relations between the methods in the target classes. The matched units are called *peer units*. Besides the included units, other feature elements of the same type in a unit are also regarded as *peer units*, such as the access flags of methods. Furthermore, to reflect the amount of information, we introduce *the length of feature* in similarity calculation, which means the number of basic elements contained in the feature. For example, the length of features of a basic block is the number of extracted instructions. Specifically, we define three types of similarity at different levels as follows:

Method-level Similarity. The proportion of the sum of the lengths of identical features to the total length of features of the method.

Class-level Similarity. The weighted average of the similarity between peer units where the weight is the length of features. If the class has nested classes, the similarity is added with the sum of the similarities of all nested classes.

Package-level Similarity. The sum of the similarity of peer units between two packages.

To support similarity calculation of packages, we propose the matching algorithm to retrieve the matching relations between classes in two packages and methods in two classes in Algorithm 2. PEDroid calculates the similarity between each two of the target units (i.e., classes or methods). It sorts the similarity scores from high to low and selects the matching pairs in turn. If the similarity of a pair is greater than THRESHOLD, the two units in the pair are considered similar. Considering the trade-off between false positives and false negatives, we set THRESHOLD as 0.15.

Algorithm 2 Matching relation construction at the class/method level.

```
Input: Members set S1, S2 in matching targets T1, T2, similarity threshold THRESHOLD
Output: Matching relationship set R
L ← ∅
for m1 in S1 do
    for m2 in S2 do
        s ← similarity between m1 and m2
        L.put(s, ⟨m1, m2⟩)
    end
end
sort L by similarity from highest to lowest
R ← ∅
for s, ⟨m1, m2⟩ in L do
    if (s > THRESHOLD) and (R have no pair containing m1 or m2) then
        R.add(⟨m1, m2⟩)
    end
end
return R
```

Positional Relationships. A package acts as the namespace, and it usually includes a collection of classes or sub-packages with similar functions. Therefore, the positional relationships between nodes in the package hierarchy indicate the relations on function. Moreover, if a subtree, consisting of a package and all its sub-packages, represents a third-party library, which is relatively independent, changes in structure generally happen within the library. Hence, two nodes with identical child nodes (or descendants) may be similar or belong to the same library.

PEDroid first retrieves candidates by three close positional relationships, i.e., the packages that have identical parent, child, or sibling packages. The nodes, which have closer relations to others, are first considered to be potentially similar. PEDroid builds the candidate collection $Candidate_{sim}$ according to the three positional relationships to identical packages in PKG_{iden}, and then selects the most similar pairs to build the matching collection PKG_{simi}.

For the nodes which cannot be matched through the close positional relationships, PEDroid obtains the similar collection $PKG^{'}_{simi}$ through the more general positional relationships in the package hierarchy, i.e., the ancestors and descendants. Algorithm 3 gives the approach to find the ancestors with matched descendants and then locate candidates by the distance to the matched ancestors. In detail, the process of matching has a loop

to search for candidates and find the most similar ones. Before the loop starts, PEDroid retrieves a set $PKG_{ancient}$ by the matched packages. It collects the node pairs having at least one matched pair in the descendant nodes. For the ith subround, PEDroid considers the nodes, whose ancestor nodes with distance i are a pair in $PKG_{ancient}$, to be candidates and adds them into $Candidate'_{sim}$. And then it obtains similar packages from $Candidate'_{sim}$ by Algorithm 1, and adds the pairs into PKG'_{simi}. Until all similar packages are found or the number of rounds exceeds the depth of the package hierarchy, the matching process is stopped.

Algorithm 3 Matching by the ancestors and descendants.

```
Input: Unmatched packages in new and old version P1,P2, two versions of hierarchy H1, H2,
       matched packages set PKG_matched
Output: Similar packages PKG'_simi
PKG_ancient ← ∅
for ⟨p1, p2⟩ in PKG_matched do
    for k = 0 .. min(level(H1, p1), level(H2, p2)) do
        a1 ← k^th ancestor of p1 in H1
        a2 ← k^th ancestor of p2 in H2
        PKG_ancient.add(⟨a1, a2⟩)
    end
end
R1, R2 ← P1, P2
PKG'_simi ← ∅
for i = 0 .. min(height(H1), height(H2)) do
    Candidate'_sim ← ∅
    for p1 in R1 do
        for p2 in R2 do
            if i > min(level(H1, p1), level(H2, p2)) then
                continue
            end
            a1 ← i^th ancestor of p1 in H1
            a2 ← i^th ancestor of p2 in H2
            if ⟨a1, a2⟩ in PKG_ancient then
                Candidate'_sim.add( ⟨p1, p2⟩)
            end
        end
    end
    matched ← get matched packages from candidate collection Candidate'_sim
    PKG'_simi.union(matched)
    for ⟨p1, p2⟩ in matched do
        R1.remove(p1)
        R2.remove(p2)
    end
end
return PKG'_simi
```

4.3 Matching Relation Extraction

With the results of package matching, PEDroid obtains matching relations (i.e. *Identical* and *Similar*) at class and method level in matched packages. The identical classes are obtained by the identical overall features of classes, while the similar classes in identical packages collected in PKG_{iden} are matched by similarity as Algorithm 2. For the similar packages in PKG_{simi} and PKG'_{simi}, the matching relations between classes have been calculated and cached during the matching process, and can be extracted directly.

Except for the matching relations, the unmatched classes/methods in the updated version of the app are classified as *New*, and those in the original version are classified as *Deleted*. Therefore, by calculating the similarity, the classes and their methods in the two packages are finally divided into four categories: *Identical*, *Similar*, *New* and *Deleted*.

5 Patch Identification

In this section, we introduce how PEDroid distinguishes whether a modified method contains a patch after locating the modified methods. Since the insight is that *a patch usually fixes the processing logic before the buggy operation or handles the errors generated by the buggy operation, while the target of operation tends to involve external values*, PEDroid analyzes the two version methods from two aspects: 1) the call sites of the methods and 2) the difference of internal semantics. Through the analysis of the call sites, PEDroid could check whether the method uses external values. Through internal semantic analysis, it locates the variables carrying external values and the original operations of these variables in the modified methods to discover potential buggy operations, and then identifies the two types of modification.

5.1 Call Site Analysis

In order to find the modified methods using external values, PEDroid employs static intra-procedural taint analysis to analyze the call sites of all modified methods. Compared with inter-procedural analysis which is more accurate but brings unacceptable overhead, the intra-procedural analysis is more suitable for us to analyze the real-world apps. And to alleviate the limitation that intra-procedural analysis cannot find external values explicitly or implicitly passed between functions, PEDroid takes the parameters and member variable as taint sources.

Since static taint analysis has been studied well, we omit its technical details for brevity here. In the following, we only describe the strategies how PEDroid selects sources and sinks and then propagates the taint.

Taint Sources. PEDroid marks the variables that may carry external values as taint sources, including parameters, member variables, and return values of method invocation statements. As a part of external values, return values of other methods are marked as sources, and external input could also be obtained by return values of Android API. Especially, the return value of the constructor method (i.e., `<init>`, `<clinit>`) without other sources is excluded for its purpose is initialization. Both the parameters and member variables could introduce external values from other methods, so PEDroid treats them as sources to avoid missing reports.

Taint Sinks. The modified methods are sinks of our taint analysis to find out whether the modified methods use external values at the call sites. PEDroid directly retrieves the methods classified as *Similar* in Section 4.3 and marks them as sinks.

Taint propagation. PEDroid mainly focuses on two types of statements, i.e., assignment and invocation, to propagate the taint.

- Assignment. If the right-hand side expression is tainted, the left-hand side value is also tainted.
- Invocation. Due to the limitation of intra-procedural analysis, it is unknown how the taint values propagate in the callee. PEDroid specifies that if a parameter is tainted, the return value and instance (if any) are also tainted, but PEDroid does not consider the possibility of taint propagation between method parameters to reduce false positives.

```
void CallerA(int arg){              void CallerB(){
    int a = this.A;                     int a = 10, b = 1;
    int b = 0;                          int c = d();
    sink(arg, a, b);                    sink(a, b, c);
}                                   }
```

Figure 3 Example for result extraction in call site analysis.

After taint propagation, PEDROID extracts the tainted states of the modified methods. For the tainted call sites, PEDROID records the indexes of all the tainted parameters and the caller. And the taint states of different call sites of a method will not be merged to reduce false positives. Figure 3 gives an example where method `sink` has two call sites in method `CallerA` and `CallerB`. In this case, PEDROID separately records that the first and second parameters of `sink` are tainted in `CallerA` and the third parameter is tainted in `CallerB`, rather than regards that all the parameters are tainted. This is because `sink` may only trigger a bug at the call site of `CallerA` and the invocation by `CallerB` has nothing to do with the bug. So, the operations of the third parameter in method `sink` can be ignored. On the other hand, `CallerB` may be a new method or the call site in `CallerB` may be newly introduced for feature enhancement. The operations of the third parameter within `sink` method are modified so that it can adapt to new features. Therefore, merging them will bring false positives.

In addition, Android callback techniques would bring false negatives to the approach, because callback methods are invoked in Android frameworks. They are driven by Android lifecycle events (e.g., `onCreate`), user interactions (e.g., `onClick`) and so on. To alleviate this problem, we collect the names of all Android callback methods in advance, and PEDROID treats the overriding callback methods as having identical call sites whose parameters are used to pass external values.

5.2 Internal Semantic Comparison

Based on the analysis of the call sites of modified methods, PEDROID identifies the patches through internal semantic comparison. Specifically, our aim is to find out whether the modification is used for correcting the processing logic or handling the errors. The former is indicated by the different dependencies of original operations, so PEDROID extracts the control and data dependencies and then compares the dependencies between two versions. As for the latter, PEDROID takes two cases into consideration. The first case is adding an exception capture operation to catch the exception generated by original operations. The second is adding checks of the return value of the original operation, while a branch of the check is a *aborting block* which aborts execution of the method when an error occurs. To identify the case, PEDROID searches for the aborting blocks by exits of methods:

1. a basic block ends with exception throwing;
2. a basic block contains only a `return` statement or logging and `return` where logging is often used to record the errors.

We implement it on the top of Soot [34]. And for illustration purpose, we take the patch in Figure 1a as example and give their Control flow graphs (CFG) in Figure 4. In detail, PEDROID compares the internal semantics through the following steps:

Step 1. Call site matching. With the modified methods and their usage, PEDROID matches the call sites between two versions to obtain all similar usage of the method in the app. Specifically, it matches the call sites whose callers have been identified as *Identical* or *Similar* in Section 4.3. According to the matching results, PEDROID analyzes each pair

```
$r0 := @this: com.Example;
```

```
$r0 := @this: com.Example;
$r1 := @parameter0: java.lang.String;
$r2 = new java.io.File;
① specialinvoke $r2.<java.io.File: void <init>(java.lang.String)>($r1);  [$r1->$r2]
② virtualinvoke $r2.<java.io.File: boolean delete()>();
return;
```

O1

(b) Buggy version of example code.

Figure 4 CFGs of the two versions of methods in Figure 1a. The example code is displayed in Soot intermediate representation. Registers in pink font indicate they depend on affected parameters, and the data flows are labeled after the statement as well. The bold statements are candidates of buggy operations.

of the call sites respectively in the following steps. It is because the matched call sites represent the identical usage of the methods and different usage should be separately analyzed as discussed in Section 5.1.

Step 2. Data flow analysis. To find usage of the tainted parameters within the method, PEDroid performs forward data flow analysis in the modified method to locate all statements which use the variables directly or indirectly dependent on these parameters. It retrieves data flows through assignment and invocation statements, where the rules are similar to propagation discussed in Section 5.1. We call the located statements *affected statements*. In Figure 4, the statements with pink registers are affected statements.

Step 3. Basic block matching. To improve the accuracy of dependency comparison, PEDroid aligns the basic blocks between the two versions of methods, instead of matching at the statement level. Alignment is based on the statements in basic blocks and the structure of CFG whose nodes are basic blocks. Due to the complexity of solving the graph matching problem, we adopt a simplified strategy that utilizes the breadth-first traversal orders of CFG to flatten the graph and aligns the blocks by LCS (longest common subsequence). The identical basic blocks are the blocks with identical representative statements including `return`, `if`, exception, method invocation, and array operations and constant values in statements.

After alignment, the blocks between two matched blocks (or entry/exit) are also regarded as matched blocks that may have many-to-many matching relations. In the example, there are three-to-one matching relationships between basic blocks which map from the basic blocks N1, N2 and N3 to O1.

And with the matching relations between basic blocks, PEDroid collects the aborting blocks which have no identical basic block. Therefore, the basic block N3 is located when analyzing the example.

Step 4. Dependency analysis. With the matching relations between basic blocks, PEDROID obtains the matched statements and then filters the subset marked in Step 2. The subset of matched statements are the original operations of the external values in the methods and includes the buggy operations we focus on. We bold these statements in the examples in Figure 4. To pinpoint which operations among the candidates (i.e., matched statements in the subset) are modified satisfying our insight, PEDROID analyzes the dependency of two types of statements.

1. To distinguish the changes to fix processing logic, PEDROID extracts control and data dependencies of each candidate in original and updated versions, which will be compared in the next step.
2. To distinguish the changes to handle errors, PEDROID analyzes the data dependency of `if` statements. Specifically, if the predecessors of the aborting blocks located in Step 3 end with a `if` statement, PEDROID searches for sources of registers compared in the statement, where the sources are the assignment statements defining these registers. If a candidate is found, PEDROID will record it as having an *error value check*. In the example, although N3 is an aborting block, the register compared is irrelevant to any candidates, so it is filtered out in this step.

Step 5. Patch identifying. Finally, PEDROID determines patches by checking two types of specific changes:

1. To check the changes for fixing the processing logic, PEDROID compares the dependencies between the original and updated methods. In particular, it compares the control and data dependencies of each candidate. A patch is reported if a difference in dependencies is found.
   In Figure 4, the candidate ① has the identical control and data dependencies between the original and updated versions, so it is not a buggy operation. But the dependencies of the candidate ② are modified where the file existence check is added in the updated version. Hence, PEDROID identifies it.
2. To check the changes for handling errors, PEDROID respectively identifies two cases. First, if an exception capture is added and its predecessors contain a candidate, it is identified as a patch. And the second case is identified by the candidate that has an error value check in the updated version but no such check in the original version.

6 Evaluation

6.1 Dataset

In the experiment, we collected two datasets, the manually selected open-source Android projects from GitHub [12] named *dBench*, and APK files of pre-installed apps extracted from Android phones. The former is used to measure the accuracy and effectiveness of PEDROID, and the latter is used to evaluate the applicability to real-world apps and check whether PEDROID can discover patches on real-world apps.

dBench: we selected apps and their updates by manually reading the commit message of the projects on GitHub, and then downloaded the release version APK files for testing, to achieve the effect on the real-world apps as far as possible. The policy for selecting updates is as follows:

1. For modification of each method in an update, detailed commits can be found so that we can determine whether a commit is used to fix a bug by the title, description, or related issue;

2. This version update has at least one patch and one non-bugfix update (e.g., code refactoring and feature enhancement). Especially, PEDroid focuses on the patches which lead to the method change and filters out other commits (e.g., configure files).

Finally, *dBench* includes 6 projects with a total of 13 updates, as shown in Table 7 and Table 8. In the tables, we also list the filtered commit IDs and whether they are marked as patches. It includes a total of 83 commits, of which 36 are marked as patches. Table 2 shows the size of APK files in each update, where the size is represented by the number of classes and methods in updated versions.

Table 2 The number of classes and methods of applications in *dBench*. ProjectName_u*n* is corresponding to each update in Table 7 and 8 for short.

Update	Classes	Methods
markor_u1	4,339	31,561
markor_u2	4,443	32,202
gpstest_u1	2,103	15,510
gpstest_u2	3,165	22,527
gpstest_u3	3,165	22,527
MaterialFiles_u1	5,822	29,637
MaterialFiles_u2	5,824	29,632
MaterialFiles_u3	7,624	42,316
andotp_u1	3,011	22,424
andotp_u2	3,996	29,155
gnucash_u1	6,688	47,398
gnucash_u2	6,690	47,414
anki_u1	14,332	135,646

Pre-installed apps: we collected pre-installed apps as a real-world app dataset. Because of the privilege permissions of pre-installed apps, the defect will lead to more serious problems. Moreover, these apps cover various categories (except games), so comprehensive types of apps can be analyzed. In detail, we collected mobile phones from six mainstream Android mobile device manufacturers, including Huawei, Motorola, Oneplus, Samsung, Vivo, and Xiaomi. In the first step, we regularly monitored app updates, and used the tool *ADB* [1] to pull the APK files from phones to the computer. For the preliminarily collected APK files, we removed duplicate files with the same hash value. Then, we used the tool *keytool* [18] to analyze the certificates of APK files, and then filtered out apps that are not signed by the vendor. Finally, the number of unique apps in our real-world dataset is 187. We regard the different APK files of an app with the minimum version gap as an update, and a total of 568 app updates are collected. The detailed amount and distribution of updated versions are shown in Table 3.

Table 3 The collected updates of pre-installed applications.

	Huawei	Motorola	Oneplus	Samsung	Vivo	Xiaomi	Total
App	42	5	25	8	28	79	187
Update	105	6	28	10	75	342	568
Major upgrade	30	1	9	0	3	34	77
Minor upgrade	16	3	6	0	19	127	171
Small update	59	2	13	10	53	181	320

6.2 Setup

Differential analysis is implemented in Python, and we disassemble the Dex bytecode of APK files by the tool *baksmali*. For patch identification, our taint analysis is based on the taint engine provided by Find Security Bugs [10], and the analysis of internal semantics is implemented in Java on top of Soot [34], a framework for analyzing and transforming Java and Android apps. In addition, PEDroid would not identify whether modified methods in the standard libraries (e.g., Android Support Library) are patches because the changes in these methods are to provide compatibility between different versions.

The experiments were performed on a server running Ubuntu 18.04 x64 with two Intel Xeon Gold 5122 Processors (each has eight logical cores at 3.60 GHz) and 128GB RAM.

6.3 Effectiveness

To measure the effectiveness of differential analysis and patch identification, we conducted a controlled experiment on *dBench*.

6.3.1 Results

In total, PEDroid found 429 modified methods which are classified as *Similar* after differential analysis and then reported 60 out of them are patches. Based on the related commits and manual analysis, the accuracy of the results will be further evaluated in Section 6.3.3 and 6.3.4. In this section, we will discuss the intermediate results and effectiveness of each phase of PEDroid.

Matching relations. 2,706 identical packages are found after identical package matching. During similar package matching, 36 packages were matched using parent-child and sibling-sibling relationships and one package was matched by ancestors and descendants. Although only one package was matched by ancestors and descendants on *dBench*, its parent package has no class to determine the similarity resulting in having no matched package, while it has no child or sibling package, so the close relationships cannot indicate the candidates for matching. Hence, matching based on ancestors and descendants is necessary for our design. In these small updates, most packages can be matched by the identical classes, and both two approaches based on positional relationships work in the process.

By class-level matching, 36,811 classes were classified as *Identical*, 251 classes were classified as *Similar*, 69 classes were classified as *New*, and 23 classes are classified as *Deleted*. Among *Similar* classes used to locate the modified methods, we found one pair of classes had the wrong matching relation. Between the two classes in the pair, a class is derived from another class in the updated version, which leads to a similar implementation and confuses matching. Unfortunately, it finally caused wrong matching relations between methods.

Modified method usage. In the call site analysis, we found a total of 1,071 call sites of *Similar* methods in updated versions, but only 893 call sites in original versions. It indicates that new call sites are introduced in the updated version of the app. Our consideration of filtering call sites in Section 5.2 is necessary.

PEDroid discovered 251 unique methods using external values by taint analysis, and 54 additional methods through the name of callback methods. We conducted a manual analysis on the filtered methods to identify false negatives. We found that most of them were filtered out because they used no external values or had no call sites (e.g., changes in the

updated third-party libraries). As for false negatives, call sites of 12 methods were missing in the taint analysis. Among them, four were overriding methods because PEDroid failed to find the correct callee at the call site, and the rest came from the lack of accuracy in the implementation of taint analysis. On the other hand, due to the limitations of callback method identification, 22 callback methods could not be found, of which three methods are customized methods by developers, and 19 methods are unrecognized due to obfuscation. In short, due to the limitations of implementation, the usage of some modified methods can not be found in analysis, most of which are caused by callbacks.

6.3.2 Performance

The time cost of each update is shown in Figure 5. PEDroid completed every analysis in 6 minutes, where taking up to 336 seconds to analyze the update `anki_u1`. According to the data in Table 2 and Figure 5, it is obvious that the time cost is greatly affected by the size of APK files. Most of the time was spent on analyzing the call sites, up to

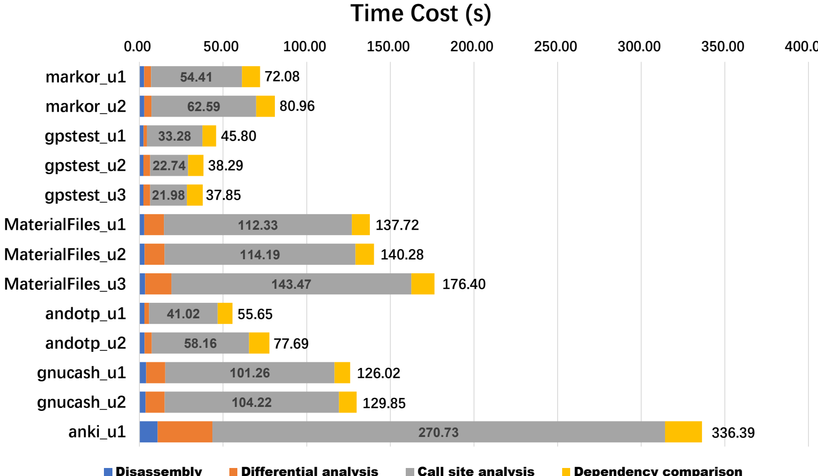


Figure 5 Time cost of each step on *dBench*.

6.3.3 Differential analysis

To evaluate the accuracy of differential analysis, we use the commits as the ground truth to check whether the modified methods are found by PEDroid. Especially, among the commits, we focus on the modifications that cause semantic changes. It means that some modifications such as renaming identifiers and merging two statements into one in commits will be ignored. In total, 238 methods have been modified by developers in *dBench*.

6.3.3.1 Accuracy

Table 4 reports the detailed results of our accuracy evaluation on *dBench*, PEDroid classified 429 methods into *Similar* category, where 234 methods belong to the project and 195 methods change with the upgrade of third-party libraries. Among the 238 modified methods, PEDroid successfully identified 221 of them, where 17 modified methods were missing. On the other hand, PEDroid mistakenly classified 13 pairs of methods as *Similar*.

It is obvious that the wrong matching relations will lead to both false negatives and false positives. For example, if two pairs $(A, A^{'})$ and $(B, B^{'})$ are modified methods, the wrong relation $(A, B^{'})$ brings a false positive and two false negatives to the results. Before illustrating the false negatives and the false positives, we conducted a manual analysis of the incorrect results and summarized the causes for wrong matching relations between methods.

Method inlining or extraction. Method inlining would merge multiple methods into one method, and extraction splits a method into multiple methods. In this case, PEDroid matches one of the methods with the highest similarity, which may wrongly match the new (or deleted) method and the long method of the other version.

Similar implementation. The implementation of some methods is very similar for their similar functions. It leads to similar extracted features, which confuse similarity calculation. When matching methods with similar implementation, the results may be crossed.

Large changes. The proportion of method body changes is large, especially for the methods with few features (e.g., only one or two basic blocks in the method body), the little change of code can lead to large changes in the extracted features. It leads to the correct matching relation can not be calculated, and the modified method is matched with irrelevant methods with partially the same features.

In the reported *Similar* methods, 13 pairs have wrong matching relations. Among them, five pairs are caused by the first reason, six pairs are caused by the second reason, and two are caused by the third reason.

The false negative refers to missing reports of modified methods. Among 17 false negatives, 13 of them are caused by wrong matching relations, which have been discussed before. Two false negatives were classified as *New* and *Deleted* by mistake due to large changes. The rest two were classified as *Identical* because the extracted features could not reflect the changes.

As for false positives, it indicates *New*/*Deleted*/*Identical* methods which are incorrectly classified as *Similar* methods, and *Similar* pairs with wrong matching relations. Especially, numbers in parentheses in Table 4 are the number of pairs with wrong matching relations. It shows that all the false positives came from the wrong matching relations.

6.3.3.2 Obfuscation-resistant

To address renaming obfuscation techniques is very important for our design. For example, the method `example()` in class `Example` was renamed with `A.a()` in the original version but `B.b()` in the updated version, which are different. Even if some of APK files in *dBench* do not enable the obfuscator, the third-party libraries it depends on are generally obfuscated. To evaluate how renaming obfuscation techniques influence apps, we counted the different method signatures (i.e., class name, method identifier, parameters, and return value of a method) between the original and updated version methods. Only in the *Similar* results, 135 of 429 *Similar* methods (31.5%) have different signatures. Moreover, based on manual analysis, only one signature is renamed by developers, and all the others are caused by compilation and obfuscation. It shows that the renaming obfuscation is commonly applied in apps, and PEDroid can resist it to a certain extent.

6.3.3.3 Comparison with previous works

We compared our approach with the previous works, including Androdiff [8], components of Androguard [3], and SimiDroid [20]. They can also provide method-level diffing between two versions of apps, and divide the results into four categories: *Identical*, *Similar*, *New* and *Deleted*. We used the same dataset *dBench* for experiment. The results are shown in Table 4.

Table 4 Comparison with Androguard and SimiDroid. The *Total* in the table indicates the number of reported methods, and the *TPL* and the *Project* indicate the reported similar methods in project source code and third-party library, respectively. The TP_P, FN_P, FP_P and $Recall_P$ indicate the accuracy in project code.

Tool	Total	TPL	Project	TP_P	FN_P	FP_P	$Recall_P$
Androdiff	816	525	291	105	133	186(16)	44.12%
SimiDroid	2111	1550	561	138	100	423(18)	57.98%
PEDroid	429	195	234	221	17	13(13)	92.86%

It is obvious that PEDroid identified much more modified methods as well retrieved less wrong matching relations, with the highest recall of 92.86%. Especially, the other two tools incorrectly regarded a large number of *Identical* methods as modified methods. Although it does not mislead patch identification, the overhead would be greatly increased. So, PEDroid is much better than the other tools.

Androdiff adopts the normalized compression distance algorithm to calculate the similarity of the two methods and extracts the instruction sequence of the basic block as the feature of the method. However, it can not resist the subtle changes caused by compilation, and most of the false positives come from the changes in the resource ID influenced by compilations. In addition, the tool does not consider the overall feature of a class and only performs similarity matching from the instructions at the method level.

SimiDroid also provides code-level similarity comparison, but it assumes that methods with identical signatures have matching relations between two versions. So, renaming obfuscation techniques have a great impact on this approach. It is the reason why SimiDroid reports much more modified methods than the other two tools, where it treats two unrelated methods as matched and detects the changes between them.

6.3.4 Patch identification

PEDroid discovered 60 patches, where 50 of them belong to the projects and 10 methods are in third-party libraries. Similar to the evaluation of differential analysis, we only evaluated the accuracy of code changes in the projects without the ground truth of third-party libraries.

6.3.4.1 Accuracy

To evaluate the accuracy of PEDroid in identifying patches, we manually identified all the patches and non-bugfix updates of all the 13 updates by analyzing their commits on GitHub. As shown in Table 7 and Table 8, among all the 83 commits in *dBench*, a total of 36 commits are identified as patch, where 47 commits are non-bugfix updates, including 35 feature updates and 12 code refactorings.

Among 36 commits containing patches, PEDroid successfully identified 28 patches during patch identification and missed eight, while it incorrectly identified seven of the 47 non-bug updates as patches. In particular, a commit could be associated with multiple modified methods. As for the amount at the method level, 41 methods were correctly identified as patches, and nine were false positives.

False negatives. The false negatives could be generally divided into three categories:

1. **Deficiency in implementation.** Four of eight false negatives come from the false negatives of call site analysis described in Section 6.3.1. It is caused by the obfuscated name of callbacks and overriding methods.
2. **Code refactoring.** We found that some patches are also accompanied by code refactoring, where the modified dependencies are encapsulated in a new method. So, PEDroid could not discover it by intra-procedural analysis, which brings two false negatives.
3. **Limitation of insight.** There are two false negatives that do not meet our insight. One is to modify the constant value in a static constructor. Another one is to add text on UI which only involves a method invocation addition without modifying any dependency.

False positives. Seven non-bugfix updates are incorrectly classified. Similarly, we also divide them into three categories:

1. **Deficiency in implementation.** One false negative comes from incorrectly matching between basic blocks. It results in different extracted dependencies at different usage of an external value.
2. **Code refactoring.** The code refactoring also leads to dependency modification, which brings two false positives to the results.
3. **Irrelevant dependency modification.** Four of the false positives are due to dependency modification irrelevant to patches. Three of them are caused by the added control dependencies, where two are to check and adapt different Android versions and one is to add a branch to enhance the feature. And the other one is introduced by the added number of parameters of the callee, which leads to the addition of data dependencies.

6.3.4.2 Comparison with other works

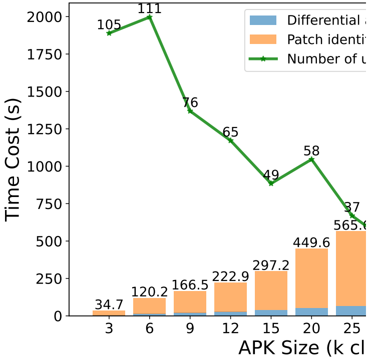
Since there is no previous work to distinguish patches from other code changes in Android apps, we evaluated whether the tool using pre-defined patterns could detect the related bugs to find out these patches. Spotbugs [35] is a state-of-the-art tool that can detect more than 400 types of bugs. Find security bugs [10] is a plugin of Spotbugs, which can detect 141 different vulnerabilities on Java and Android apps.

First, we applied *dBench* on the tool SpotBugs with its component Find Security Bugs, and detected the original and updated versions of the app updates respectively. Then we found out the difference of the bug reports between two versions with the method-level matching relations generated by differential analysis. Finally, only two different bug reports were found, and they belonged to one commit. It is because detecting bugs according to manually defined patterns has limitations which cannot discover the unknown bugs.

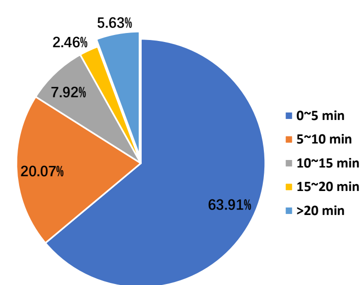
6.4 Applicability

6.4.1 Performance

PEDroid extracted a total number of 98,591 patches from the dataset. In detail, 45,805 patches were identified in 320 small updates, 31,549 patches were identified in 171 minor upgrades and 21,237 patches were identified in 77 major upgrades. The time cost is shown in Figure 6a, where the updates are grouped by the size of APK files (e.g., the first group consists of updates with the number of classes less than 3000, and so on). It shows that size of apps has a great impact on the overhead of PEDroid, especially for patch identification. Since the number of updates in each group is different, Figure 6a also gives the number.



(a) Time cost of analyzing updates w



(b) Distribution of time cost of analyzing updates.

Figure 6 Performance on real world dataset.

Furthermore, the time cost distribution of updates is given in Figure 6b. It is concluded that 63.91% of updates could be analyzed within 5 minutes, 83.98% of apps could be analyzed within 10 minutes, and 94.37% could be analyzed within 20 minutes.

6.4.2 Analysis of Extracted Patches

In order to illustrate that PEDroid can help the analysis based on patches, we made a further analysis to understand the patches extracted from updates of the pre-installed apps.

6.4.2.1 Discovered Patches

To demonstrate that PEDroid can extract effective patches from the real-world apps, we first randomly selected several reports on pre-installed apps for manual analysis. We discovered many typical cases of patches, and the security check addition appears most among them, which confirms the conclusion of the previous work [41]. Another common repair case is adding an exception-capture operation to prevent the app from crashing. In this section, we discuss the typical cases and how they improve the security and stability of apps.

Security check. Adding security checks is a common way to fix bugs. This type of patch can be detected because a new control dependency is always added. Due to complex scenarios such as network communication, local data access, and user interaction, the added security check also has various purposes, where two of the most common cases are checking whether the referenced object is null to avoid `NullPointerException`, and calling `TextUtils.isEmpty` to prevent empty strings. In addition, we show two typical cases of adding black and white list checks to discuss the security improvement by checking addition.

Figure 7(a) gives a patch with a white list check. The method has `@JavascriptInterface` annotation, which means that it can be invoked by web pages in WebView. In the fixed version of the method, the domain name of the web page which invokes this method is checked, and only the domain names in the white list are allowed to use this method, which increases the security.

(a) white list

```
  @JavascriptInterface
  public void caseWhiteList(String arg){
+         if(!DomainUtils.checkUrl(WebActivity->mUrl,
                                WebActivity.WHITE_LIST_JS)){
+             return;
+         }
      ...
  }
```

(b) black list

```
  public class Case{
      private boolean caseBlackList(DiskShowItem arg8, List arg9) {
          if(!arg8.mIsDir) {
+             if(FileUtils.checkSpecialChars(arg8.mFileName) >= 0) {
+                 return true;
+             }
              ...
          }
      }
  }

  public final class FileUtils {
      static{
          FileUtils.specialChars = new char[]{'\\', '/', ':', '*', '?',
                                            '\"', '<', '>', '|', '\n'};
          ...
      }
      public static int checkSpecialChars(String arg5) {
          if(!TextUtils.isEmpty(arg5)) {
              char[] v0 = FileUtils.specialChars;
              for(int v2 = 0; v2 < v0.length; ++v2) {
                  char v3 = v0[v2];
                  if(arg5.indexOf(v3) >= 0) {
                      return arg5.indexOf(v3);
                  }
              }
          }
          return -1;
      }
  }
```

(c) data processing

```
  public void dataProcessing() {
      ...
      String[] dirs = getDirectoryPathList();
      for(int i=0; i<dirs.length; i++){
-         File file = new File(dirs[i], "lib.so");
+         File file = new File(dirs[i].trim(), "lib.so");
          ...
      }
      ...
  }
```

(d) field addition to record status

```
  protected void onDestroy() {
      super.onDestroy();
+     if(this.mHasBinded) {
          this.unbindService(this.mServiceConnection);
+         this.mHasBinded = false;
+     }
  }
```

```
  public class CaseClass {
      static {
-         CaseClass.CRPYT_IV_BYTE = new byte[]{34, 0x20, 33, … , 35, 0x20, 0x20};
-         CaseClass.CRPYT_KEY_BYTE = new byte[]{33, 34, 35, … , 35, 34, 33};
          ...
      }
      public CaseClass(Context arg2) {
          ...
          this.mCryptoUtil.init(this.mContext);
+         this.CRPYT_IV_BYTE = this.mCryptoUtil.initIV();
+         this.CRPYT_KEY_BYTE = this.mCryptoUtil.initKey();
          this.loadData();
      }
      protected void loadData() {
          ...
-         String iv = Cryptor.xorKey(Case3Class.CRPYT_IV_BYTE);
-         String key = Cryptor.xorKey(Case3Class.CRPYT_KEY_BYTE);
+         String iv = Cryptor.xorKey(this.CRPYT_IV_BYTE);
+         String key = Cryptor.xorKey(this.CRPYT_KEY_BYTE);
          String data = new String(Cryptor.decrypt(iv, key, Base64.decode(cipher, 0)), "utf-8");
          ...
      }
  }
```

Figure 8 Case Study for hard-coded key removal.

The function of the method in Figure 7(b) is to download files. The security check at line 3 is added to resolve a vulnerability. The method `checkSpecialChars` checks whether there are special characters in the file name. The existence of these special characters could lead to path traversal vulnerability. Once these special characters are detected, this method returns directly and does not continue downloading the target file.

Data processing. Figure 7(c) gives an example of modification of data dependencies to correct data processing. In the buggy version, the blank characters are not trimmed after obtaining the path of the directory. As a result, the corresponding library cannot be found and the function is unavailable. This patch will be reported through modification of data dependencies extracted from the invocation of the constructor of `File`.

Field addition for status recording. This patch is applied to check before resource access or release and sets the field to the corresponding value when resources are required and released. The case is found through the inconsistency of control dependencies. The case is shown in Figure 7(d).

Hard-coded key removal. A security patch of discarding the usage of hard-coded keys is given in Figure 8. The decryption key and IV used in the original version are hard-coded and defined in the static constructor (`<clinit>`). The updated version is generated in the constructor (`<init>`). PEDroid identified the patch by comparing dependencies between the two versions of the method `loadData`. In the buggy version, the hard-coded key and IV are static member variables of the class, and its acquisition has nothing to do with the affected parameter `this`. But in the fixed version, the decryption key and IV are generated at runtime, which are bound to the object instance, and have a data dependency on the parameter `this` which uses external values.

In addition to the examples of modifying the processing logic listed above, handling the errors is also commonly encountered in our manual analysis, including the error value check to end wrong execution and exception capture to prevent crashes. Since these cases are easy to understand, we would not list them here. Especially, exception capture will be further discussed later.

6.4.2.2 Application of Patches

Based on the typical patches, we further identified similar patches to find out what patches are frequently applied to fix bugs and whether the developers make the mistakes commonly. Specifically, we selected the five simple patch cases found in the manual analysis and used the buggy and fixed versions of the method and the potential buggy operations in reports to determine whether the patch is the same type as the cases. For security checks, we collected two common types, i.e., the addition of `null` and `TextUtils.isEmpty` check before the buggy operation. And we located the added invocation of `trim` which was used to correct the data processing of a buggy operation. Similarly, when a check of a `boolean` field is added and the state of the field is modified around the buggy operation, the check would be marked as field addition for status recording. For exception capture, we focused not only on the addition of exception capture but also on the types of exceptions.

Table 5 shows the usage of different types of common patches in all the extracted patches. It is reported that the check of null reference is added most commonly, similar to the results of our manual analysis. Even if we only searched a simple case of correcting data processing (i.e., string trimming), we still found that several developers at different vendors, made the same mistake and repaired it. It shows that it is a feasible means to summarize the problems that have been repaired to find similar problems in other apps.

Table 5 Usage of common patches in updates.

Type	Total
Null Reference	7682
Empty String	1409
Status Record	269
String Trimming	23
Exception	6289

Table 6 Top 10 most common types of added exception catching.

Type	Total
Ljava/lang/Exception	3838
Ljava/lang/Throwable	1353
Ljava/io/IOException	1212
Ljava/lang/IllegalArgumentException	663
Lorg/json/JSONException	633
Ljava/lang/RuntimeException	457
Ljava/lang/NumberFormatException	284
Ljava/lang/IllegalStateException	234
Ljava/lang/IllegalAccessException	225
Ljava/lang/SecurityException	223

In addition, we analyzed exception-capture patches and found the types of exceptions that are easily ignored during development. Table 6 gives the top 10 most common types among our extracted patches and the number of exception-capture patches corresponding to

each type. Especially, a patch could add the capture of multiple types of exceptions at the same time, so the exception-capture patches counted in Table 5 may be counted multiple times in Table 6. It shows that developers often simply use the basic type `Exception` to catch all types of exceptions, as well `Throwable` which can catch both exceptions and errors. As for other types of exceptions, the capture of `IOException` is patched most frequently in the extracted patches because it can be thrown by unexpected behaviors in a variety of scenarios including network and file I/O. The exceptions are easy to be accidentally missed by developers.

7 Discussion

7.1 Limitation and Future works

In the following, we discuss limitations and future works to improve the accuracy of the analysis performed by PEDroid.

First, PEDroid is designed to resist the renaming obfuscation because it has been broadly used by many Android applications. However, to be sensitive to code changes and efficiently retrieve matching relations, PEDroid chooses to retain features of instructions in the method body and utilizes package trees to assist the matching process. Given our current design, some advanced obfuscations can impede PEDroid to a certain degree. For example, some obfuscation tools can move a sub-package from one package to another, so as to modify the package hierarchy. Considering commonly-used obfuscators such as ProGuard do not totally break package structures, and our approach does not require the package structures to be exactly identical, we believe the selected strategies are acceptable in practice.

Second, PEDroid is mainly designed based on static intra-procedural analysis considering applicability to real-world apps. However, only analyzing the data dependencies and original operations within a single method could bring both false positives and false negatives, especially when meeting code refactoring. Meanwhile, the more precise usage of external values is more likely obtained through the inter-procedural taint analysis. We believe the inter-procedural feature could be implemented by considering method invocation, which is an interesting future work.

Third, PEDroid tries to find out patches and the corresponding bugs without manually defined patterns [19] or generated signatures of known patches or bugs [44]. Although the approach could not cover patches of all types of bugs (e.g., the two false negatives beyond the insight), it could make up for the gap in this research field to a certain degree. And we have evaluated the effectiveness by running our approach on *dBench*, and identified most patches. The results on the real-world dataset also show that rich types of bugs can be discovered through this approach.

7.2 Usage of Extracted Patches

In the paper, we discovered some typical cases of bugs and patches in Android apps and summarized the rules by manually analyzing the patches to distinguish them. Similarly, several APR (Automated Program Repair) techniques adopt manually defined code transformation schema to automatically repair bugs in Android apps [48, 25, 42, 5, 36]. Therefore, it is feasible to summarize new schemas through the analysis of the extracted patches and then apply them to APR. In addition, lots of efforts focus on learning from the existing patches which require no manually defined templates and empirical knowledge [17, 40, 26, 24, 37, 21]. However, these works are all designed for repairing source code rather than bytecode. We believe that our work can make up for the lack of learning data sets to promote the proposal of the technique on bytecode.

The extracted patches can also be used to detect similar bugs. Some binary-level similarity detection and code reuse detection techniques [15, 46] can take the buggy version of patched methods as the comparison target and detect whether there are similar problems in other apps.

8 Conclusion

We propose an approach to extract bytecode-level patches from Android apps, which includes two phases: obtaining the modified methods from the neighboring versions of Android apps and identifying patches among them. To achieve the first step and resist name-based obfuscation, we employ similarity comparison at the method level based on code features and the structure of the app. We design an approach to detect patches by analyzing the usage and internal semantics of the original and updated versions of methods. We applied the approach to extract patches from 13 updates of open-source projects and identified 28/36 patches. To evaluate the applicability to real-world apps, we further performed an experiment on the real-world dataset, which is proved that this approach can find various types of patches within a reasonable amount of time.

References

1 Android debug bridge (adb), accessed: November 2021. URL: https://developer.android.com/studio/command-line/adb.

2 Open source two-factor authentication for android, accessed: November 2021. URL: https://github.com/andOTP/andOTP.

3 androguard, accessed: November 2021. URL: https://code.google.com/archive/p/androguard/.

4 Ankidroid: Anki flashcards on android. your secret trick to achieve superhuman information retention, accessed: November 2021. URL: https://github.com/ankidroid/Anki-Android.

5 Tanzirul Azim, Iulian Neamtiu, and Lisa M. Marvel. Towards self-healing smartphone software via automated patching. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 623–628. ACM, 2014.

6 Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 356–367. ACM, 2016.

7 Bindiff, accessed: November 2021. URL: https://www.zynamics.com/bindiff.html.

8 Anthony Desnos. Android: Static analysis using similarity distance. In *45th Hawaii International International Conference on Systems Science (HICSS-45 2012), Proceedings, 4-7 January 2012, Grand Wailea, Maui, HI, USA*, pages 5394–5403. IEEE Computer Society, 2012.

9 Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

10 Find security bugs, accessed: November 2021. URL: https://find-sec-bugs.github.io/.

11 git-difftool documentation, accessed: November 2021. URL: https://git-scm.com/docs/git-difftool.

12 Github: Where the world builds software, accessed: November 2021. URL: https://github.com/.

13 Gnucash for android mobile companion application, accessed: November 2021. URL: `https://github.com/codinguser/gnucash-android`.

14 open-source android gnss/gps test program, accessed: November 2021. URL: `https://github.com/barbeau/gpstest`.

15 Steve Hanna, Ling Huang, Edward XueJun Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, volume 7591 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 2012.

16 Project planning for developers, accessed: November 2021. URL: `https://github.com/features/issues`.

17 Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309. ACM, 2018.

18 keytool, accessed: November 2021. URL: `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html`.

19 Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 802–811. IEEE Computer Society, 2013.

20 Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*, pages 136–143. IEEE Computer Society, 2017.

21 Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 602–614. ACM, 2020.

22 Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.*, 3(OOPSLA):162:1–162:30, 2019.

23 Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 118–129. IEEE Computer Society, 2018.

24 Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 727–739. ACM, 2017.

25 Siqi Ma, David Lo, Teng Li, and Robert H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 711–722. ACM, 2016.

26 Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, volume 10493 of *Lecture Notes in Computer Science*, pages 229–246. Springer, 2017.

27 Text editor - notes & todo (for android), accessed: November 2021. URL: `https://github.com/gsantner/markor`.

28 Material design file manager for android, accessed: November 2021. URL: `https://github.com/zhanghai/MaterialFiles`.

29 Stuart McIlroy, Nasir Ali, and Ahmed E. Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empir. Softw. Eng.*, 21(3):1346–1370, 2016.

30 Shrink your java and android code, accessed: November 2021. URL: `https://www.guardsquare.com/proguard`.

31 Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *International Conference on Software Engineering (ICSE)*, pages 471–480, New York, NY, USA, 2008. ACM.

32 Danilo Silva, João Paulo da Silva, Gustavo Jansen de Souza Santos, Ricardo Terra, and Marco Tulio Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Trans. Software Eng.*, 47(12):2786–2802, 2021.

33 Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 269–279. IEEE Press, 2017.

34 Soot – A java optimization framework, accessed: November 2021. URL: `https://github.com/soot-oss/soot`.

35 Spotbugs, accessed: November 2021. URL: `https://spotbugs.github.io/`.

36 Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 187–198. ACM, 2018.

37 Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

38 Xinda Wang, Kun Sun, Archer L. Batcheller, and Sushil Jajodia. Detecting "0-day" vulnerability: An empirical study of secret security patch in OSS. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 485–492. IEEE, 2019.

39 Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. ORLIS: obfuscation-resilient library detection for android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018*, pages 13–23. ACM, 2018.

40 Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 479–490. IEEE, 2019.

41 Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020.* The Internet Society, 2020.

42 Jiayun Xie, Xiao Fu, Xiaojiang Du, Bin Luo, and Mohsen Guizani. Autopatchdroid: A framework for patching inter-app vulnerabilities in android application. In *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–6. IEEE, 2017.

43 Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 54–65. ACM, 2005.

44 Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proc. 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Virtual Event, USA, 2020. ACM.

45 Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 462–472. IEEE / ACM, 2017.

46 Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. Detecting java code clones with multi-granularities based on bytecode. In *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1*, pages 317–326. IEEE Computer Society, 2017.

47 Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. Libid: reliable identification of obfuscated third-party android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 55–65. ACM, 2019.

48 Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

49 Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 141–152. IEEE Computer Society, 2018.

A Dataset

A.1 *dBench*

dBench includes six popular open source Android apps on GitHub shown as Table 7 and 8. Except for *markor* with 900+ stars, other projects have 1k-4.4k stars.

Table 7 Updates in *dBench* and all commits – part.1.

Project	Old Version	New Version	Commit id	Bug fix
markor[27]	2.2.3	2.2.5	5b53574c88888ecbcc4b5c712d26a4c0e4f89650	✗
			464579b59047bbacb2f9fb7edb9fb9563a9dfe2c	✗
			35e25bff0de3521a41c4574561b958a8068fafa1	✗
			d0a5103223430e7af925a48f49affa0ae64ef83b	✗
			37a9c135e7a2502f8ce1b6b463614a7c10168816	✓
			9dd83708e49f45d85e2c4f3ef9cc21a3019d327d	✗
			cbd37234b587222c974b29a196f54c8f20f08b77	✗
			14cd95d37d0c12bacb2bd290bdee07d4a949ea24	✓
			47cff19dd5030d2c3ce470ce525fb2ab20f19727	✗
			22b7681cb52eb4f820c1bd036683b102be144b82	✗
			57745bb82ef225223e6780f65bc0d5dabf81cead	✓
	2.3.1	2.3.3	11895e5554c59033927a7fb5e8139797165a703d	✓
			e182dcc64057cd5f1bd8ac63492de4fa6f2f6658	✓
			51e8febed782e824ae4953bc266777828afc076e	✓
			2f5352c59e8e1edc15ad7825d3b50d0980ec70b1	✗
			46d9165b0a6f3a6a6e243fb2e8c4417c9bab0666	✓
			c9a9cc7736084355cc422b3822a8da61d58b9569	✓
			d24f2cb29d76422d5e01f69d9b01b1ff78c8c8db	✗
			63808c166aef82aaee2ed5ca67dd8a10eb2fa054	✗
			df02630b66914176f28d07a32ccde9478d20742c	✗
			6e2b07c7c1b61718904096245f9106fd14b1447e	✗
			f725a85011fc9342d37f55c58ba35926a94b6d0a	✗
			76184b2aa73a215d7e5c66a3dfee6db8f8cfad1d	✗
			27a0e8506abcdcaf2d7801493712eafb4e6ffbd7	✗
gpstest[14]	3.7.4	3.8.0	0b47fca1a9f06017b6d319269764ac6cec9b1f7b	✓
			8ed5b31c8e356b79cfe8b8bba49a10156101f758	✗
	3.9.5	3.9.6	c14a1025d6026aebef5747fb53eb28e891b02501	✓
			944733d36f44451096823200242f0ebdd5ef02c6	✓
			396c52a796e924cc5507bb087b4eadd684806fda	✗
	3.9.6	3.9.7	70d8ec5197117660e6251945e804829e5221dbbe	✓
			5625b632c4a60767950f61651629d09c8cb9fbe2	✗
MaterialFiles[28]	1.0.0-beta.11	1.0.0-rc.1	b864874d87450591f20562f1e240ff228393c554	✗
			cfcfce564e42db79a7668dbedab978a35dd01e1e	✓
			e0f488a7950402ac6464dae451b7a462898af316	✗
			8480642ddf39521eff7f30a79c5d1feec5a7d4cb	✓
			2c379913b0cf6272e1b60da265a3f7ab32cfdaaf	✗
			0d98dc34fc1cee5908514aa8eb8679f82c3d36dc	✓
	1.0.0	1.0.1	fdd9940d98974b8291496922ddb98714162b0ccc	✗
			041d384eed4cdc85d16ef063dd966a300b3b4769	✗
			428fab2cb24512e90d6d94e781134e85de29c104	✓
			fbc862d8a80bca16365dd8cfc42f0f846b0b2935	✓
			c81f380f4ec11071f139f3993987b15d3cb4a77c	✓
			4b14cefb59d746822e1f31a92ecf46e15c2d88ff	✗
	1.2.0	1.2.1	a5c07bc764c0678d423594ff454349ab63def5aa	✗
			fc22c3ada63c8392b1dcced1c96d818404ba140b	✓
			b78c799aa0f356d551c12904f07e2c9dfd3aba8e	✗
			0f0d306e5db2e2afea257449c050936c5a60a5c0	✓
			d4918e0c5a3e11d0f7e49033aa3625c5b5138da9	✗
			618806bafcf6cc424b84471d485744f96dba4b4b	✓
			ac8ca9988f761b5e8cdf7d0ecbd47d215540d145	✓

Table 8 Updates in *dBench* and all commits – part.2.

Project	Old Version	New Version	Commit id	Bug fix
andOTP[2]	0.2.6	0.2.7	77655b610897eb59e6ff7fcc4f13454f34b4a86d	✗
			f0518a265c858414b74ef84e2e8bd945a96ad59a	✗
			dd97ac87f059f8c1498d17d7c99ac6dc70068ea5	✗
			f41eb620aadb3dd203f923d934ce1f6da713c901	✗
			cbdced2df1d5ab5fd35d17c7230b60a89d3d4012	✗
			247f4e938ed6def7668e3259c81a6fc9e1dd5db0	✗
			842d49b68f86412d246c9ab9a8d59dcbc11c4f8c	✗
			ce696861c7497a67c72be0a315fc9d1e5cbd0489	✓
	0.7.1	0.7.1.1	73f8c14ec389a2ad8c2a61edef2bcfd4b4894b70	✓
			cdc54028b3395401fa65665bc5e01e6a279071d3	✗
			c1d6c6b2b8c01fbfb3a0ab7ba5b3c247bf80cd3f	✗
			5215308a1afcf7744998509674507252 01dbb1c9	✗
gnucash-android[13]	2.1.1	2.1.2	57241e8c064302a215aa74501e0dc1ba31e6a096	✓
			1794882757a37c108c4b4cf40f6876aa7a51c87d	✓
			dae1caf7078bdd3e425e25cbfd5a37eb2309e0e6	✓
			f81ad6067a4136b34ccfc277cd21913682a3ce31	✓
			a363eebaff01f7fdadbda5edc661aa35133a450a	✗
			404759620a5a33cecf0bf836fe5802401eacf4d6	✓
	2.1.2	2.1.3	ff894a5ce5901bafc8626279d09278efc229ef23	✗
			6048bd8d0604370a38189dad9ba451aa121fc7bb	✓
			a6aa211734accf94664da91316cf6e26bed0de92	✓
			b2e9bf7f38a287985656e48ec6b13979a070dcd0	✗
			d790b805ec17fd22ab4566ae1d24cefe72486e36	✓
			724a686177798685112a02fcc3873873fb7a9595	✓
			952cb2b697b9bd946437e19db4597d23b3446f55	✓
Anki-Android[4]	2.16alpha24	2.16alpha25	a38503e08c0a8f0445adb527a015aa3a82cd4404	✗
			672c44eb664284339b697bff27ec8b37925c3c31	✓
			5135b06f4ca61cb15f75973362e2d25340925524	✗
			09430ad55c4186f5d9e528480059652703603 08d	✗
			81d1d134863b8ab2c0560f9f11148b6a91996c0d	✓
			99ea713f780a428332990d3e5b7033d714a3ffad	✗
			b7d283f96fd3922806beb5eeb499e475f034d5a8	✗
			0f7b0bebed9539c6ee46608539be23c2e5db4780	✗

Ferrite: A Judgmental Embedding of Session Types in Rust

Ruo Fei Chen
Independent Researcher, Leipzig, Germany

Stephanie Balzer
Carnegie Mellon University, Pittsburgh, PA, USA

Bernardo Toninho
NOVA LINCS, Nova University Lisbon, Portugal

Abstract

Session types have proved viable in expressing and verifying the protocols of message-passing systems. While message passing is a dominant concurrency paradigm in practice, real world software is written without session types. A limitation of existing session type libraries in mainstream languages is their restriction to linear session types, precluding application scenarios that demand sharing and thus aliasing of channel references. This paper introduces Ferrite, a shallow embedding of session types in Rust that supports both *linear* and *shared* sessions. The formal foundation of Ferrite constitutes the shared session type calculus SILL_S, which Ferrite encodes via a novel *judgmental embedding* technique. The fulcrum of the embedding is the notion of a typing judgment that allows reasoning about shared and linear resources to type a session. Typing rules are then encoded as functions over judgments, with a valid typing derivation manifesting as a well-typed Rust program. This Rust program generated by Ferrite serves as a *certificate*, ensuring that the application will proceed according to the protocol defined by the session type. The paper details the features and implementation of Ferrite and includes a case study on implementing Servo's canvas component in Ferrite.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Type theory; Software and its engineering → Domain specific languages; Software and its engineering → Concurrent programming languages

Keywords and phrases Session Types, Rust, DSL

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.22

Related Version *Technical Report*: https://arxiv.org/abs/2009.13619 [7]

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.14

Funding *Stephanie Balzer*: National Science Foundation Award No. CCF-1718267.
Bernardo Toninho: FCT/MCTES grant NOVALINCS/BASE UIDB/04516/2020.

1 Introduction

Message-passing is a dominant concurrency paradigm, adopted by mainstream languages such as Erlang, Scala, Go, and Rust, putting the slogan *"Do not communicate by sharing memory; instead, share memory by communicating"* [13] into practice. In this setting, messages are exchanged along channels, which can be shared by several senders and receivers. Type systems in such languages typically allow channels to be typed, specifying and constraining the types of messages they may carry (e.g. integers, strings, sums, references, etc.).

An aspect inherent to message-passing concurrency that is not captured in mainstream type systems, however, is the idea of a *protocol*. Protocols dictate the sequencing and types of messages to be exchanged. To express and enforce such protocols, *session types* [14, 15, 16]

© Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho;
licensed under Creative Commons License CC-BY 4.0

were introduced. Session typing disciplines assign types to channel endpoints according to their intended usage protocols in terms of sequencing of input/output actions (e.g. "send an integer and, afterwards, receive a string") and branching/selection actions (e.g. "receive either a buy message and process the payment; or a cancellation message and abort the transaction"), ensuring the action sequence is followed correctly and thus, adherence to the protocol. Thanks to their correspondence to *linear* logic [4, 44, 43, 42, 26, 5] session types enjoy a strong logical foundation and ensure, in addition to protocol adherence (*session fidelity*), the existence of a communication partner (*progress*). Session types have also been extended with safe *sharing* [1, 2, 3] to accommodate multi-client scenarios that are rejected by exclusively linear session types.

Despite these theoretical advances, session types have not (yet) been adopted at scale. While various session type embeddings exist in mainstream languages such as Java [18, 17], Scala [39], Haskell [38, 34, 20, 27], OCaml [32, 19], and Rust [21, 25, 8, 9], all of these embeddings lack support for multi-client scenarios that mandate controlled aliasing in addition to linearity.

This paper introduces *Ferrite* [6], a shallow embedding of session types in Rust. In contrast to prior work, Ferrite supports *both* linear and shared session types, with protocol adherence guaranteed statically by the Rust compiler. Ferrite's underlying theory is based on the calculus $\mathsf{SILL_S}$ introduced in [1], which develops the logical foundation of shared session types. As a matter of fact, Ferrite encodes $\mathsf{SILL_S}$ typing derivations as Rust functions, through a technique we dub *judgmental embedding*. Through our judgmental embedding, a type-checked Ferrite program yields a Rust program that corresponds to a $\mathsf{SILL_S}$ typing derivation and thus the *proof* of protocol adherence.

In order to faithfully encode $\mathsf{SILL_S}$ typing in Rust, this paper further makes several technical contributions to emulate advanced typing features, such as higher-kinded types, by a skillful combination of traits (type classes) and associated types (type families). For example, Ferrite supports recursive (session) types in this way, which are limited to recursive structs of a fixed size in plain Rust. A combination of type-level natural numbers with ideas from profunctor optics [33] are also used to support named channels and labeled choices. We adopt the idea of *lenses* [11] for selecting and updating individual channels in an arbitrary-length linear context. Similarly, we use *prisms* for selecting a branch out of arbitrary-length choices. Whereas `session-ocaml` [32] has previously explored the use of n-ary choice through extensible variants in OCaml, we are the first to connect n-ary choice to prisms and non-native implementation of extensible variants. Notably, the Ferrite codebase remains entirely in the *safe* fragment of Rust, with no (direct) use of unsafe features.

Given its support of both linear and shared session types, Ferrite is capable of expressing any session-typed program in Rust. We substantiate this claim by providing an implementation of Servo's canvas component with the communication layer within Ferrite.

This work makes the following contributions: **(i)** the design and implementation of *Ferrite*, an embedded domain-specific language (EDSL) for writing session-typed programs in Rust; **(ii)** with support of both *linear* and *shared* sessions, guaranteed to be observed by type checking; **(iii)** a novel *judgmental embedding* of custom typing rules in a host language with the resulting program carrying the proof of successful type checking; **(iv)** an encoding of *arbitrary-length choice* in terms of prisms and extensible variants in Rust; **(v)** an *empirical evaluation* based on a full implementation of Servo's canvas component in Ferrite.

All typing rules and their encoding as well as further materials of interest to an inquisitive reader are provided in our companion technical report [7].

Table 1 Overview of session types and terms in $\mathsf{SILL_S}$ together with their operational meaning. Subscripts L and S denote linear and shared sessions, resp., where $m, n \in \{\mathsf{L}, \mathsf{S}\}$.

Session type		**Process term**		
current	**cont**	**current**	**cont**	**Description**
$c_{\mathsf{L}}{:}\oplus\{\overline{l{:}A_{\mathsf{L}}}\}$	$c_{\mathsf{L}}{:}A_{\mathsf{L}_h}$	$c_{\mathsf{L}}.l_h; P$	P	provider sends label l_h along c_{L}
		$\mathsf{case}\ c_{\mathsf{L}}\ \mathsf{of}\ \overline{l \Rightarrow Q}$	Q_h	client receives label l_h along c_{L}
$c_{\mathsf{L}}{:}\&\{\overline{l{:}A_{\mathsf{L}}}\}$	$c_{\mathsf{L}}{:}A_{\mathsf{L}_h}$	$\mathsf{case}\ c_{\mathsf{L}}\ \mathsf{of}\ \overline{l \Rightarrow P}$	P_h	provider receives label l_h along c
		$c_{\mathsf{L}}.l_h; Q$	Q	client sends label l_h along c_{L}
$c_{\mathsf{L}}{:}A_m \otimes B_{\mathsf{L}}$	$c_{\mathsf{L}}{:}B_{\mathsf{L}}$	$\mathsf{send}\ c_{\mathsf{L}}\ d_m; P$	P	provider sends channel $d_m{:}A_m$ along c_{L}
		$y_m \leftarrow \mathsf{recv}\ c_{\mathsf{L}}; Q_{y_m}$	Q_{d_m}	client receives channel $d_m{:}A_m$ along c_{L}
$c_{\mathsf{L}}{:}A_m \multimap B_{\mathsf{L}}$	$c_{\mathsf{L}}{:}B_{\mathsf{L}}$	$y_m \leftarrow \mathsf{recv}\ c_{\mathsf{L}}; P_{y_m}$	P_{d_m}	provider receives channel $d_m{:}A_m$ along c_{L}
		$\mathsf{send}\ c_{\mathsf{L}}\ d_m; Q$	Q	client sends channel $d_m{:}A_m$ along c_{L}
$c_{\mathsf{L}}{:}\mathbf{1}$	-	$\mathsf{close}\ c_{\mathsf{L}}$	-	provider sends "end" along c_{L}
		$\mathsf{wait}\ c_{\mathsf{L}}; Q$	Q	provider receives "end" along c_{L}
$c_{\mathsf{L}}{:}\downarrow^{\mathsf{S}}_{\mathsf{L}}A_{\mathsf{S}}$	$c_{\mathsf{S}}{:}A_{\mathsf{S}}$	$c_{\mathsf{S}} \leftarrow \mathsf{detach}\ c_{\mathsf{L}}; P_{x_{\mathsf{S}}}$	$P_{c_{\mathsf{S}}}$	provider sends "$\mathsf{detach}\ c_{\mathsf{S}}$" along c_{L}
		$x_{\mathsf{S}} \leftarrow \mathsf{release}\ c_{\mathsf{L}}; Q_{x_{\mathsf{S}}}$	$Q_{c_{\mathsf{S}}}$	client receives "$\mathsf{detach}\ c_{\mathsf{S}}$" along c_{L}
$c_{\mathsf{S}}{:}\uparrow^{\mathsf{S}}_{\mathsf{L}}A_{\mathsf{L}}$	$c_{\mathsf{L}}{:}A_{\mathsf{L}}$	$c_{\mathsf{L}} \leftarrow \mathsf{acquire}\ c_{\mathsf{S}}; Q_{x_{\mathsf{L}}}$	$Q_{c_{\mathsf{L}}}$	client sends "$\mathsf{acquire}\ c_{\mathsf{L}}$" along c_{S}
		$x_{\mathsf{L}} \leftarrow \mathsf{accept}\ c_{\mathsf{S}}; P_{x_{\mathsf{L}}}$	$P_{c_{\mathsf{L}}}$	provider receives "$\mathsf{acquire}\ c_{\mathsf{L}}$" along c_{S}
$c_m : A_m$	$c_m : A_m$	$z_n \leftarrow X \leftarrow \overline{d_m}; P_{z_n}$	P_{z_n}	spawn ("cut") X along $z_n{:}B_n$ with $\overline{d_m{:}D_m}$
$c_m : A_m$	-	$\mathsf{fwd}\ c_m\ d_m$	-	forward to channel $d_m{:}A_m$ and terminate

2 Background

This section gives a brief tour of linear and shared session types. The presentation is based on the intuitionistic session-typed process calculus $\mathsf{SILL_S}$ [1], which Ferrite builds upon. We consider the protocol governing the interaction between a queue and its client:

$$\mathsf{queue}\ A = \&\{\mathsf{enq} : A \multimap \mathsf{queue}\ A,\ \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : A \otimes \mathsf{queue}\ A\}\}$$

Table 1 provides an overview of the types used in the example. Since $\mathsf{SILL_S}$ is based on a Curry-Howard correspondence between intuitionistic linear logic and the session-typed π-calculus [4, 5] it uses linear logic connectives ($\oplus$, $\&$, $\otimes$, $\multimap$, $\mathbf{1}$) as session types. The remaining connectives concern shared sessions, a feature we remark on shortly. A crucial – and probably unusual – characteristic of session-typed processes is that a process *changes* its typing along with the messages it exchanges. As a result, a process' typing always reflects the current protocol state. Table 1 lists state transitions inflicted by a message exchange in the first and second column and corresponding process terms in the third and fourth column. The fifth column provides the operational meaning of a type.

Consulting Table 1, we gather that the above polymorphic session type $\mathsf{queue}\ A$ imposes the following recursive protocol: A client may either send the label enq or deq to the queue, depending on whether the client wishes to enqueue or dequeue an element of type A, resp. In the former case, the client sends the element to be enqueued, after which the queue recurs. In the latter case, the queue indicates to the client whether it is empty (none) or not (some), and proceeds by either terminating or sending the dequeued element and recurring, resp.

A linear typing discipline is beneficial because it immediately guarantees session fidelity – even in the presence of perpetual protocol change – by ensuring that a channel connects exactly two processes. Unfortunately, linearity also rules out various practical programming scenarios that demand sharing and thus aliasing of channel references. For example, the above linear session type $\mathsf{queue}\ A$ is limited to a *single* client. To support safe sharing of

stateful channel references while upholding session fidelity, $\mathsf{SILL_S}$ extends linear session types with shared session types ($\downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}$, $\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$). These two connectives mediate between shared and linear sessions by requiring that clients of shared sessions interact in *mutual exclusion* from each other. Concretely, a type $\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$ mandates a client to *acquire* the process offering the shared session. If the request is successful, the client receives a linear channel to the acquired process along which it must proceed as detailed by the session type A_{L}. A type $\downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}$, on the other hand, mandates a client to *release* the linear process, relinquishing ownership of the linear channel and only being left with a shared alias to the now shared process at type A_{S}.

Using these connectives, we can turn the above linear queue into a shared one, bracketing enqueue and dequeue operations within acquire-release:

$$\mathsf{squeue}\ A_{\mathsf{S}} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \& \{\mathsf{enq} : A_{\mathsf{S}} \multimap \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{squeue}\ A_{\mathsf{S}},\ \mathsf{deq} : \oplus\{\mathsf{none} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{squeue}\ A_{\mathsf{S}},\ \mathsf{some} : A_{\mathsf{S}} \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{squeue}\ A_{\mathsf{S}}\}\}$$

In contrast to the linear queue, the above version recurs in the none branch and thus keeps the queue alive to serve the next client. For convenience, $\mathsf{SILL_S}$ allows the connectives $\otimes$ and $\multimap$ to be used to transport both linear and shared channels along a linear carrier channel.

To provide a flavor of session-typed programming in $\mathsf{SILL_S}$, we briefly comment on the below processes *empty* and *elem*, which implement the shared queue session type as a sequence of *elem* processes, ended by an *empty* process. A process implementation consists of its signature (first two lines) and body (after =). The first line indicates the typing of channel variables used by the process (left of $\vdash$) and the type of the providing channel variable (right of $\vdash$). The second line binds the channel variables. In $\mathsf{SILL_S}$, $\leftarrow$ generally denotes variable bindings. We leave it to the reader to convince themselves, consulting Table 1, that the code in the body of the two processes executes the protocol defined by session type $\mathsf{squeue}\ A_{\mathsf{S}}$.

$$
\begin{array}{l}
\cdot \vdash \mathit{empty} :: q : \mathsf{squeue}\ A_{\mathsf{S}} \\
q \leftarrow \mathit{empty} \leftarrow \cdot = \\
\quad q' \leftarrow \mathsf{accept}\ q\ ; \\
\quad \mathsf{case}\ q'\ \mathsf{of} \\
\quad \mid \mathsf{enq} \rightarrow x \leftarrow \mathsf{recv}\ q'\ ; \\
\qquad\qquad q \leftarrow \mathsf{detach}\ q'\ ; \\
\qquad\qquad e \leftarrow \mathit{empty}\ ; q \leftarrow \mathit{elem} \leftarrow x, e \\
\quad \mid \mathsf{deq} \rightarrow q'.\mathsf{none}\ ; \\
\qquad\qquad q \leftarrow \mathsf{detach}\ q'\ ; \\
\qquad\qquad q \leftarrow \mathit{empty}
\end{array}
$$

$$
\begin{array}{l}
x : A_{\mathsf{S}}, t : \mathsf{squeue}\ A_{\mathsf{S}} \vdash \mathit{elem} :: q : \mathsf{squeue}\ A_{\mathsf{S}} \\
q \leftarrow \mathit{elem} \leftarrow x, t = \\
\quad q' \leftarrow \mathsf{accept}\ q\ ; \\
\quad \mathsf{case}\ q'\ \mathsf{of} \\
\quad \mid \mathsf{enq} \rightarrow y \leftarrow \mathsf{recv}\ q'\ ; \\
\qquad\qquad t' \leftarrow \mathsf{acquire}\ t\ ; \\
\qquad\qquad t'.\mathsf{enq}\ ; \mathsf{send}\ t'\ y\ ; \\
\qquad\qquad t \leftarrow \mathsf{release}\ t'\ ; q \leftarrow \mathsf{detach}\ q'\ ; \\
\qquad\qquad q \leftarrow \mathit{elem} \leftarrow x, t \\
\quad \mid \mathsf{deq} \rightarrow q'.\mathsf{some}\ ; \mathsf{send}\ q'\ x\ ; \\
\qquad\qquad q \leftarrow \mathsf{detach}\ q'\ ; \mathsf{fwd}\ q\ t
\end{array}
$$

Imposing acquire-release not only as a programming methodology but also as a *typing discipline* has the advantage of recovering session fidelity for shared sessions. To this end, shared session types in $\mathsf{SILL_S}$ must be *strictly equi-synchronizing* [1, 3], imposing the invariant that an acquired session is released to the type at which previously acquired. For example, the shared session type $\mathsf{squeue}\ A_{\mathsf{S}}$ is strictly equi-synchronizing whereas the type $\mathsf{invalid} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \& \{\mathsf{left} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \uparrow^{\mathsf{S}}_{\mathsf{L}} \oplus \{\mathsf{yes} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{invalid}, \mathsf{no} : \mathbf{1}\}, \mathsf{right} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{invalid}\}$ is not.

It is instructive to review the typing rules for acquire-release:

$$\text{(T-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\text{L)}\quad \frac{\Psi, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}; \Delta, y_{\mathsf{L}} : A_{\mathsf{L}} \vdash Q_{y_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Psi, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}; \Delta \vdash y_{\mathsf{L}} \leftarrow \mathsf{acquire}\, x_{\mathsf{S}}\, ; Q_{y_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}$$

$$\text{(T-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\text{R)}\quad \frac{\Psi; \cdot \vdash P_{y_{\mathsf{L}}} :: (y_{\mathsf{L}} : A_{\mathsf{L}})}{\Psi \vdash y_{\mathsf{L}} \leftarrow \mathsf{accept}\, x_{\mathsf{S}}\, ; P_{y_{\mathsf{L}}} :: (x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}})}$$

$$\text{(T-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\text{L)}\quad \frac{\Psi, x_{\mathsf{S}} : A_{\mathsf{S}}; \Delta \vdash Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Psi; \Delta, y_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}} \vdash x_{\mathsf{S}} \leftarrow \mathsf{release}\, y_{\mathsf{L}}\, ; Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}$$

$$\text{(T-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\text{R)}\quad \frac{\Psi \vdash P_{x_{\mathsf{S}}} :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Psi; \cdot \vdash x_{\mathsf{S}} \leftarrow \mathsf{detach}\, y_{\mathsf{L}}\, ; P_{x_{\mathsf{S}}} :: (y_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}})}$$

Due to its foundation in intuitionistic linear logic, $\mathsf{SILL_S}$' typing rules are phrased using a *sequent calculus*, leading to *left* and *right* rules for each connective. Left rules describe the interaction from the point of view of the client, right rules from the point of view of the provider. The typing judgments $\Psi; \Delta \vdash P :: (x_{\mathsf{L}} : A_{\mathsf{L}})$ and $\Psi \vdash P :: (x_{\mathsf{S}} : A_{\mathsf{S}})$ read as "process P offers a session of type A along channel x using sessions offered along channels in Ψ (and Δ)." The typing contexts Ψ and Δ provide the typing of shared and linear channels, resp. Whereas Ψ is a structural context, Δ is a linear context, forbidding channels to be dropped (weakened) or duplicated (contracted). In contrast to linear processes, shared processes must not use any linear channels, a requirement crucial for type safety. The notions of acquire and release are naturally formulated from the point of view of a client, so these terms appear in the left rules. The right rules use the terms *accept* and *detach* with the meaning that an accept accepts an acquire and a detach initiates a release. The rules are read bottom-up, where the premise denotes the next action to be taken after the message exchange.

3 Key Ideas

This section introduces the key ideas underlying Ferrite. Subsequent sections provide further details.

3.1 $\mathsf{SILL_R}$ – A stepping stone from $\mathsf{SILL_S}$ to Ferrite

In Section 2, we reviewed $\mathsf{SILL_S}$ and its typing judgment. Our goal with Ferrite is to faithfully and compositionally encode $\mathsf{SILL_S}$ typing derivations in Rust. However, when viewed under the lens of a general purpose programming language, most readers will find $\mathsf{SILL_S}$ a prohibitively austere formalism, lacking most facilities needed to write realistic programs (e.g. basic data types, pattern matching, etc.) and provided by a convenient and usable programming language like Rust. From an ergonomics standpoint alone it would be unreasonably prohibitive for our embedding to forbid the use of Rust features such as functions, traits and enumerations, only for the sake of precisely mirroring $\mathsf{SILL_S}$. Moreover, to realize such an embedding we must be able to account for both $\mathsf{SILL_S}$' linear session discipline (i.e. the *linear* context Δ) and shared session discipline (i.e. the *structural* context Ψ) within Rust's usage discipline. Since Rust's typing discipline is essentially *affine*, its treatment of variable usage is neither linear nor purely structural, and so both shared and linear channels must be treated explicitly in the encoding.

The two points above naturally lead us to the language $\mathsf{SILL_R}$ as a formal stepping stone between $\mathsf{SILL_S}$ and our embedding, Ferrite. $\mathsf{SILL_R}$ is, in its essence, a pragmatic extension of $\mathsf{SILL_S}$ with Rust (type and term) constructs, allowing us to intersperse Rust code with the communication primitives of $\mathsf{SILL_S}$. In $\mathsf{SILL_R}$ we use the judgment $\Gamma; \Delta \vdash \mathit{expr} :: A$, denoting that expression *expr* has session type A, using the sessions tracked by Γ and Δ.

This judgment differs from that of $\mathsf{SILL_S}$ in its context region Γ and term *expr*, with the latter permitting arbitrary Rust expressions in addition to $\mathsf{SILL_S}$ primitives. Whereas $\mathsf{SILL_S}$'s structural context Ψ exclusively tracks shared channels, $\mathsf{SILL_R}$'s Γ tracks *both* shared sessions (subject to weakening and contraction) and plain Rust (affine) variables. A shared channel type in both $\mathsf{SILL_R}$ and $\mathsf{SILL_S}$ is always of the form $\uparrow_{\mathsf{L}}^{\mathsf{S}} A$, so there is no confusion among the affine and shared contents of Γ. As we discuss in Section 5.2, the distinction between a plain Rust variable, which is treated as affine, and a shared channel, which is treated structurally, is modelled in Ferrite by making shared channels implement Rust's `Clone` trait.

Table 2 Overview of $\mathsf{SILL_R}$ types and terms and their encoding in Ferrite. Note that $\mathsf{SILL_R}$ uses $\tau \triangleleft A_\mathsf{L}$ and $\tau \triangleright A_\mathsf{L}$ for shared channel output and input, resp., and ϵ for termination.

Type Ferrite	$\mathsf{SILL_R}$	**Terms** ($\mathsf{SILL_R}$) provider	client
`InternalChoice<Row>`	$\oplus\{\overline{l_i : A_{\mathsf{L}_i}}\}$	offer $l_i; K$	case a $\{\overline{l_i : K_i}\}$
`ExternalChoice<Row>`	$\&\{\overline{l_i : A_{\mathsf{L}_i}}\}$	offer_choice$\{\overline{l_i : K_i}\}$	choose a $l_i; K$
`SendChannel<A,B>`	$A_\mathsf{L} \otimes B_\mathsf{L}$	send_channel_from $a; K$	$a \leftarrow$ receive_channel_from f $a; K$
`ReceiveChannel<A,B>`	$A_\mathsf{L} \multimap B_\mathsf{L}$	$a \leftarrow$ receive_channel; K	send_channel_to f $a; K$
`SendValue<T,A>`	$\tau \triangleleft A_\mathsf{L}$	send_value $x; K$	$x \leftarrow$ receive_value_from a $x; K$
`ReceiveValue<T,A>`	$\tau \triangleright A_\mathsf{L}$	$x \leftarrow$ receive_value; K	send_value_to a $x; K$
`End`	ϵ	terminate	wait $a; K$
`SharedToLinear<A>`	$\downarrow^\mathsf{S}_\mathsf{L} A_\mathsf{S}$	detach_shared_session; K_s	release_shared_session $a; K_l$
`LinearToShared<A>`	$\uparrow^\mathsf{S}_\mathsf{L} A_\mathsf{L}$	accept_shared_session; K_l	$a \leftarrow$ acquire_shared_session $s; K_l$

Table 2 provides an overview of $\mathsf{SILL_R}$ types and terms and their Ferrite encoding. $\mathsf{SILL_R}$ types stand in direct correspondence with $\mathsf{SILL_S}$ types (see Table 1), apart from shared channel output and input. The $\mathsf{SILL_S}$ types for sending and receiving shared channels ($A_\mathsf{S} \otimes A_\mathsf{L}$ and $A_\mathsf{S} \multimap A_\mathsf{L}$) correspond to $\mathsf{SILL_R}$ types for sending and receiving values ($T \triangleleft A$ and $T \triangleright A$, resp.), which support *both* Rust values and shared channels. Their typing rules are:

$$(\mathrm{T}\triangleleft_\mathsf{R})\quad \frac{\Gamma\,;\,\Delta \vdash K :: A}{\Gamma,\, x:\tau;\,\Delta \vdash \textsf{send_value}\ x;\ K :: \tau \triangleleft A}$$

$$(\mathrm{T}\triangleleft_\mathsf{L})\quad \frac{\Gamma,\, x:\tau\,;\Delta, a:A \vdash K :: B}{\Gamma\,;\Delta, a:\tau \triangleright A \vdash x \leftarrow \textsf{receive_value_from}\ a;\ K :: B}$$

Rule $\mathrm{T}\triangleleft_\mathsf{R}$ indicates that the value bound to variable x of type τ will be sent, after which the continuation K will execute, offering type A. Dually, rule $\mathrm{T}\triangleleft_\mathsf{L}$ states that using such a provider bound to a will bind x of type τ in continuation K, which must now use the channel bound to a according to A.

3.2 Judgmental Embedding

Having introduced the $\mathsf{SILL_R}$ typing judgment and illustrated some of its typing rules, we can now clarify the idea behind our notion of *judgmental embedding*, which enables the Rust compiler to typecheck $\mathsf{SILL_R}$ programs by encoding typing derivations as Rust programs. The basic idea underlying this encoding can be schematically described as follows:

$$\frac{\Gamma\,;\Delta_2 \vdash \mathit{cont} :: A_2}{\Gamma\,;\Delta_1 \vdash \mathit{expr};\mathit{cont} :: A_1}$$

```
fn expr<...>
  ( cont: PartialSession<C2, A2> )
  -> PartialSession<C1, A1>
```

On the left we show a $\mathsf{SILL_R}$ typing rule and on the right its encoding in Ferrite. Ferrite encodes a $\mathsf{SILL_R}$ *typing judgment* $\Gamma;\Delta \vdash \mathit{expr} :: A$ as a value of Rust *type* `PartialSession<C, A>`, where `C` encodes the linear context Δ and `A` the session type A, standing for any of the Ferrite types of Table 2. Ferrite then encodes a $\mathsf{SILL_R}$ *typing rule* for an expression *expr* as a Rust *function* `expr` that accepts a `PartialSession<C2, A2>` and returns a `PartialSession<C1, A1>`, where *expr* stands for any of the $\mathsf{SILL_R}$ terms of Table 2. The encoding makes use of *continuation passing style* (arising from the sequent calculus-based formulation of $\mathsf{SILL_R}$), with the return type being the conclusion of the rule and the argument type being its premise. Table 3 summarizes the judgmental embedding; Section 4.1 provides further details. Whereas Ferrite explicitly performs a type-level encoding of the linear context Δ, the representation of the shared and affine context region Γ is achieved through Rust's normal

Table 3 Judgmental embedding of $\mathsf{SILL_R}$ in Ferrite.

$\mathsf{SILL_R}$	**Ferrite**	**Description**
$\Gamma\,;\cdot \vdash A$	`Session<A>`	Typing judgment for top-level session (i.e. closed program).
$\Gamma\,;\Delta \vdash A$	`PartialSession<C, A>`	Typing judgment for partial session.
Δ	`C: Context`	Linear context; explicitly encoded.
Γ	-	Shared / Affine context; delegated to Rust.
A	`A: Protocol`	Session type.

binding structure, with the obligation that shared channels implement Rust's `Clone` trait to permit contraction. To type a closed program, Ferrite defines the type `Session<A>`, which stands for a $\mathsf{SILL_R}$ judgment with an empty linear context.

Adopting a judgmental embedding technique for implementing a DSL delivers the benefits of proof-carrying code: the `PartialSession<C1, A1>` returned from a well-typed Ferrite `expr` *is* the typing derivation of the corresponding $\mathsf{SILL_R}$ term. In case the $\mathsf{SILL_R}$ term is a $\mathsf{SILL_S}$ term, its typing derivation certifies protocol adherence by virtue of the type safety proof of $\mathsf{SILL_S}$ [1]. In case the $\mathsf{SILL_R}$ term includes Rust code, its typing derivation certifies protocol adherence modulo the possibility of a panic raised by the Rust code. A fully general type safety result for $\mathsf{SILL_R}$, possibly building upon existing formalizations of Rust [22], is an avenue of future work.

3.3 Recursive and Shared Session Types in Ferrite

Rust's support for recursive types is limited to recursive struct definitions of a known size. To circumvent this restriction and support arbitrary recursive session types, Ferrite introduces a type-level fixed-point combinator `Rec<F>` to obtain the fixed point of a type function `F`. Since Rust lacks higher-kinded types such as `Type` $\rightarrow$ `Type`, we use *defunctionalization* [36, 46] by accepting any Rust type `F` implementing the trait `RecApp` with a given associated type `F::Applied`, as shown below. Section 5.1 provides further details.

```
trait RecApp<X> { type Applied; }
struct Rec<F: RecApp<Rec<F>>> { unfold: Box<F::Applied> }
```

Recursive types are also vital for encoding shared session types. In line with [3], we restrict shared session types to be recursive, making sure that a shared component is continuously available. To guarantee type preservation, recursive session types must be *strictly equi-synchronizing* [1, 3], requiring an acquired session to be released to the same type at which it was previously acquired. Ferrite enforces this invariant by defining a specialized trait `SharedRecApp` which omits an implementation for `End`:

```
trait SharedRecApp<X> { type Applied; }   trait SharedProtocol { ... }
struct SharedToLinear<F> { ... }    struct SharedChannel<S: SharedProtocol> { ... }
struct LinearToShared<F: SharedRecApp<SharedToLinear<LinearToShared<F>>>> { ... }
```

Ferrite achieves safe communication for shared sessions by imposing an acquire-release discipline [1] on shared sessions, establishing a critical section for the linear portion of the process enclosed within acquire and release. `SharedChannel` denotes the shared process running in the background, and clients with a reference to it can *acquire* an exclusive linear channel to communicate with it. As long as the linear channel exists, the shared process is locked and cannot be acquired by any other client. With the strictly equi-synchronizing constraint in place, the now linear process must eventually be released (`SharedToLinear`) back to the same shared session type at which it was previously acquired, giving turn to another client waiting to acquire. Section 5.2 provides further details on the encoding.

3.4 N-ary Choice and Linear Context

Ferrite implements n-ary *choices* and linear typing *contexts* as extensible *sums* and *products* of session types, resp. Ferrite uses heterogeneous lists [23] to annotate a list of session types of arbitrary length. The notation HList!$[A_0, A_1, ..., A_{N-1}]$ denotes a heterogeneous list of N session types, with A_i being the session type at the i-th position of the list. The HList! macro acts as syntactic sugar for the heterogeneous list, which in its raw form is encoded as $(A_0, (A_1, (..., (A_{N-1}, ()))))$. Ferrite uses the Rust tuple constructor `(,)` for HCons, and unit `()` for HNil. The heterogeneous list itself can be directly used to represent an n-ary product. Using an associated type, the list can moreover be transformed into an n-ary sum.

One disadvantage of using heterogeneous lists is that its elements have to be addressed by position rather than a programmer-chosen label. To recover labels for accessing list elements, we use optics [33]. More precisely, Ferrite uses *lenses* [11] to access a channel in a linear context and *prisms* to select a branch of a choice. We further combine the optics abstraction with *de Bruijn levels* and implement lenses and prisms using type level natural numbers. Given an inductive trait definition of natural numbers as zero (`Z`) and successor (`S<N>`), a natural number `N` implements the lens to access the N-th element in the linear context, and the prism to access the N-th branch in a choice. Schematically, the lens encoding can be captured as follows:

$$\frac{\Gamma\,;\Delta,\,l_n : B_2 \;\vdash\; K :: A_2}{\Gamma\,;\Delta,\,l_n : B_1 \;\vdash\; \mathit{expr}\;l_n; K :: A_1}$$

```
fn expr<...>
  ( l: N, cont: PartialSession<C1, A2> )
  -> PartialSession<C2, A1>
where N: ContextLens<C1, B1, B2, Target=C2>
```

The index `N` amounts to the type of the variable `l` that the programmer chooses as a name for a channel in the linear context. Ferrite handles the mapping, supporting random access to programmer-named channels. Section 4.2 provides further details, including the support of higher-order channels. Similarly, prisms allow choice selection in constructs such as `offer_case` to be encoded as follows:

$$\frac{\Gamma; \Delta \;\vdash\; K :: A_n}{\Gamma; \Delta \;\vdash\; \textsf{offer_case}\; l_n;\; K :: \oplus\{..., l_n : A_n, ...\}}$$

```
fn offer_case<N, Row, C, A>
  ( l: N, cont: PartialSession<C, A> )
  -> PartialSession<C, InternalChoice<Row>>
where N: Prism<Row, Elem=A>, ...
```

Ferrite maps a choice label to a constant having the singleton value of a natural number `N`, which implements the prism to access the N-th branch of a choice. In addition to prisms, Ferrite implements a version of *extensible variants* [28] to support polymorphic operations on arbitrary sums of session types representing choices. Finally, the `define_choice!` macro is used as a helper to export type aliases as programmer-friendly identifiers. Details are reported in Section 6 and in our companion technical report [7].

4 Ferrite – A Judgmental Embedding of SILL_R

Having introduced some of the key concepts underlying the implementation of Ferrite, we now cover in detail the implementation of Ferrite's core constructs, building up the knowledge required for Section 5 and Section 6. Ferrite, like any other DSL, has to tackle the various technical challenges encountered when embedding a DSL in a host language. In doing so, we take inspiration from the range of embedding techniques developed for Haskell and adjust them to the Rust setting. The lack of higher-kinded types, limited support of recursive types, and presence of weakening, in particular, make the development far from trivial. A more conceptual contribution of this work is thus to demonstrate how existing Rust features can be combined to emulate many of the missing features that are beneficial to DSL embeddings

and how to encode custom typing rules in Rust or any similarly expressive language. The techniques described in this and subsequent sections also serve as a reference for embedding other DSLs in a host language like Rust.

4.1 Encoding Typing Rules via Judgmental Embedding

A distinguishing characteristic of Ferrite is its *propositions as types* approach, yielding a direct correspondence between $\mathsf{SILL_R}$ notions and their Ferrite encoding. This correspondence was introduced in Section 3.2 (see Table 3) and we now discuss it in more detail. To this end, let's consider the typing of value input. We remind the reader of Table 2 in Section 3, which provides a mapping between $\mathsf{SILL_R}$ and Ferrite session types. Interested readers can find a corresponding mapping on the term level in the companion technical report [7].

$$\frac{\Gamma,\, a:\tau\,;\Delta \;\vdash\; K \;::\; A}{\Gamma\,;\Delta \;\vdash\; a \leftarrow \mathsf{receive_value};\; K \;::\; \tau \triangleright A}\;(\mathrm{T}\triangleright_{\mathsf{R}})$$

The $\mathsf{SILL_R}$ right rule $\mathrm{T}\triangleright_{\mathsf{R}}$ types expression $a \leftarrow \mathsf{receive_value};\, K$ with session type $\tau \triangleright A$ and the continuation K with session type A, where a is now in scope with type τ. Following the schema hinted in Section 3.2, Ferrite encodes this rule as the function `receive_value`, parameterized by a value type `T` (τ), a linear context `C` (Δ), and an offered session type `A`.

```
fn receive_value<T, C:Context, A:Protocol>(cont:impl FnOnce(T) -> PartialSession<C, A>)
                                           -> PartialSession<C, ReceiveValue<T, A>>
```

The function yields a value of type `PartialSession<C, ReceiveValue<T, A>>`, i.e. the conclusion of the rule, given an (affine) closure of type `T` $\rightarrow$ `PartialSession<C, A>`, encoding the premise of the rule. Notably, Ferrite uses plain Rust binding (through function types) to encode the contents of Γ, as illustrated for the received value above. The use of a closure reveals the continuation-passing-style of the encoding, where the received value of type `T` is passed to the continuation closure. The affine closure implements the `FnOnce` trait, ensuring that it can only be called once.

The type `PartialSession` is a core construct of Ferrite that enables the judgmental embedding of $\mathsf{SILL_R}$. A Rust value of type `PartialSession<C, A>` represents a Ferrite program that guarantees linear usage of session type channels in the linear context `C` and offers the linear session type `A`, corresponding to the $\mathsf{SILL_R}$ typing judgment $\Gamma; \Delta \vdash \mathit{expr} :: A$. The type parameters `C` and `A` are constrained to implement the traits `Context` and `Protocol` – two other Ferrite constructs representing a linear context and linear session type, resp.:

```
trait Context { ... }     trait Protocol { ... }
struct PartialSession<C: Context, A: Protocol> { ... }
```

For each $\mathsf{SILL_R}$ session type, Ferrite defines a corresponding Rust struct that implements the trait `Protocol`, yielding the listing shown in Table 2. Implementations for ϵ (`End`) and $\tau \triangleright A$ (`ReceiveValue<T, A>`) are shown below. When a session type is nested within another session type, such as in the case of `ReceiveValue<T, A>`, the constraint to implement `Protocol` is propagated to the inner session type, requiring `A` to also implement `Protocol`:

```
struct End { ... }    struct ReceiveValue<T, A> { ... }
impl Protocol for End { ... }  impl<A: Protocol> Protocol for ReceiveValue<T, A> { ... }
```

Thus, while Ferrite delegates the handling of the shared/structural context Γ to Rust, the encoding of the linear context Δ is explicit. Being affine, the Rust type system permits weakening, a structural property rejected by linear logic. Ferrite encodes a linear context as a heterogeneous (type-level) list [23] of the form `HList![`$\mathtt{A}_0$`, `$\mathtt{A}_1$`, ..., `$\mathtt{A}_{N-1}$`]`, with all its type

elements A_i implementing `Protocol`. Internally, the `HList` macro desugars the type-level list into a nested tuple `(A0, (A1, (..., (AN−1, ()))))`. The unit type `()` is used as the empty list (`HNil`) and the tuple constructor `(,)` is used as the `HCons` constructor. The implementation for `Context` is defined inductively as follows:

```
impl Context for () { ... }   impl<A: Protocol, C: Context> Context for (A, C) { ... }
```

To represent a closed program, i.e. a program without free channel variables, we define a type alias `Session<A>` for `PartialSession<C, A>`, with `C` restricted to the empty context:

```
type Session<A> = PartialSession<(), A>;
```

A complete session type program in Ferrite is thus of type `Session<A>` and amounts to the $\mathsf{SILL_R}$ typing derivation proving that the program adheres to the defined protocol. Below we show a "hello world"-style program in Ferrite:

```
let hello_provider = receive_value(|name| { println!("Hello, {}", name); terminate() });
```

The Ferrite program `hello_provider` has an inferred Rust type `Session<ReceiveValue<String, End>>`. It offers the type `ReceiveValue<String, End>` by first receiving a string value using `receive_value`, binding it to `name` in the continuation closure. Upon receiving the name string, It prints out the name with a `"Hello"` greeting, and terminates using `terminate()`.

4.2 Manipulating the Linear Context

Context Lenses

The use of a type-level list to encode the linear context has the advantage of allowing contexts of arbitrary length. However, the list imposes an order on the context's elements, disallowing exchange. To allow exchange, we make use of the concept of *lenses* [11] to define a `ContextLens` trait, which is implemented using type-level natural numbers.

```
#[derive(Copy)] struct Z;     #[derive(Copy)] struct S<N>(PhantomData<N>);
trait ContextLens<C: Context, A1: Protocol, A2: Protocol> { type Target: Context; ... }
```

The `ContextLens` trait defines the read and update operations on a linear context, such that given a *source* context `C = HList![..., AN, ...]`, the source element of interest, A_N at position N, can be updated to the target element `B` to form the *target* context `Target = HList![..., B, ...]`, with the remaining elements unchanged. We use natural numbers to inductively implement `ContextLens` at each position in the linear context, such that it satisfies all constraints of the form:

```
N: ContextLens<HList![..., AN, ...], AN, B, Target=HList![..., B, ...]>
```

The implementation of natural numbers as context lenses is done by first considering the base case, with `Z` used to access the first element of any non-empty linear context:

```
impl<A1: Protocol, A2: Protocol, C: Context> ContextLens<(A1, C), A1, A2>
  for Z { type Target = ( A2, C ); ... }
impl<A1: Protocol, A2: Protocol, B: Protocol, C: Context, N: ContextLens<C, A1, A2>>
 ContextLens <(B, C), A1, A2> for S<N> { type Target = (B, N::Target); ... }
```

In the inductive case, for any natural number `N` implementing the context lens for a context `HList![A0, ..., AN, ...]`, it's successor `S<Z>` implements the context lens for `HList![A−1, A0, ..., AN, ...]`, with a new element A_{-1} appended to the head of the linear context. Using context lenses, we can encode the $\mathsf{SILL_R}$ left rule $\mathrm{T}\triangleright_{\mathsf{L}}$ shown below, which types sending an ambient value x to a channel a in the linear context that expects to receive a value.

$$\frac{\Gamma\,;\,\Delta, a : A \vdash K :: B}{\Gamma,\, x : \tau;\, \Delta,\, a : \tau \triangleright A \vdash \mathsf{send_value_to}\; a\; x;\; K :: B}\;(\mathrm{T}{\triangleright}_{\mathsf{L}})$$

In Ferrite, $\mathrm{T}{\triangleright}_{\mathsf{L}}$ is implemented as the function `send_value_to`, which uses a context lens `N` to send a value of type `T` to the `N`-th channel in the linear context `C1`. This requires the `N`-th channel to have type `ReceiveValue<T,A>`. A continuation `cont` is then given with the linear context `C2`, which has the `N`-th channel updated to type `A`.

```
fn send_value_to<N, T, C1: Context, C2: Context, A: Protocol, B: Protocol>
  ( n: N, x: T, cont: PartialSession<C2, B> ) -> PartialSession <C1, B>
where N: ContextLens<C1, ReceiveValue<T, A>, A, Target=C2>
```

Channel Removal

The above definition of a context lens is suited for *updating* channel types in a context. However, we have not addressed how channels can be *removed* or *added* to the linear context. These operations are required to implement session termination and higher-order channel constructs such as $\otimes$ and $\multimap$. To support channel removal, we introduce a special `Empty` element to denote the *absence* of a channel at a given position in the linear context:

```
struct Empty;      trait Slot { ... }
impl Slot for Empty { ... }     impl<A: Protocol> Slot for A { ... }
```

To allow `Empty` to be present in a linear context, we introduce a new `Slot` trait and make both `Empty` and `Protocol` implement `Slot`. The original definition of `Context` is then updated to allow types that implement `Slot` instead of `Protocol`.

$$\frac{\Gamma\,;\,\Delta \vdash K :: A}{\Gamma\,;\,\Delta,\; a : \epsilon \vdash \mathsf{wait}\,a;\; K :: A}\;(\mathrm{T1}_{\mathsf{L}}) \qquad \frac{}{\Gamma\,;\,\cdot \vdash \mathsf{terminate};\, :: \epsilon}\;(\mathrm{T1}_{\mathsf{R}})$$

Using `Empty`, it is straightforward to implement $\mathsf{SILL}_{\mathsf{R}}$'s session termination. Rule $\mathrm{T1}_{\mathsf{L}}$ is encoded via a context lens that replaces a channel of session type `End` with the `Empty` slot. The function `wait` shown below does not really remove a slot from a linear context, but merely replaces the slot with `Empty`. The use of `Empty` is necessary, because we want to preserve the position of channels in a linear context in order for the context lens for a channel to work across continuations.

```
fn wait<C1: Context, C2: Context, A: Protocol, N>
  ( n: N, cont: PartialSession<C2, A> ) -> PartialSession<C1, A>
where N: ContextLens<C1, End, Empty, Target=C2>
```

With `Empty` introduced, an empty linear context may now contain any number of `Empty` slots (e.g., `HList![Empty, Empty]`). We introduce an `EmptyContext` trait to abstract over the different forms of empty linear contexts and provide an inductive definition as its implementation:

```
trait EmptyContext: Context { ... }   impl EmptyContext for () { ... }
impl<C: EmptyContext> EmptyContext for (Empty, C) { ... }
```

Given the empty list `()` as the base case, the inductive case `(Empty, C)` is an empty linear context, if `C` is also an empty linear context. Using the definition of an empty context, the $\mathsf{SILL}_{\mathsf{R}}$ right rule $\mathrm{T}\mathbf{1}_{\mathsf{R}}$ can then be easily encoded as the function `terminate`, which works generically for all contexts that implement `EmptyContext` as shown below:

```
fn terminate<C: EmptyContext>() -> PartialSession<C, End>
```

Channel Addition

The Ferrite function `wait` removes a channel from the linear context by replacing it with `Empty`. Dually, the function `receive_channel`, adds a new channel to the linear context. The $\mathsf{SILL_R}$ rule $\mathrm{T}{\multimap}_{\mathsf{R}}$ for channel input is shown below. It binds the received channel of session type A to the channel variable a and adds it to the linear context Δ of the continuation.

$$\frac{\Gamma\,;\,\Delta, a : A \vdash K :: B}{\Gamma\,;\,\Delta \vdash a \leftarrow \mathsf{receive_channel};\, K :: A \multimap B}\ (\mathrm{T}{\multimap}_{\mathsf{R}})$$

To encode $\mathrm{T}{\multimap}_{\mathsf{R}}$, an append operation on contexts is defined via the `AppendContext` trait:

```
trait AppendContext<C: Context>: Context { type Appended: Context; ... }
impl<C: Context> AppendContext<C> for () { type Appended = C; ... }
impl<A: Slot, C1: Context, C2: Context, C3: Context> AppendContext<C2>
  for (A, C1) where C1: AppendContext<C2, Appended=C3> { type Appended = (A, C3); ... }
```

The `AppendContext` trait is parameterized by a linear context `C` and an associated type `Appended`. If a linear context `C1` implements the trait `AppendContext<C2>`, it means that context `C2` can be appended to `C1`, with `C3` = `C1::Appended` being the result of the append operation. The implementation of `AppendContext` is defined inductively, with the empty list `()` implementing the base case and the cons cell `(A, C)` implementing the inductive case.

Using `AppendContext`, a channel `B` can be appended to the end of a linear context `C`, if `C` implements `AppendContext<HList![B]>`. The new linear context after the append operation is given in the associated type `C::Appended`. We then observe that the position of channel `B` in `C::Appended` is the same as the length of the original linear context `C`. In other words, the context lens for channel `B` in `C::Appended` can be generated by obtaining the length of `C`. In Ferrite, the length operation is implemented by adding an associated type `Length` to the `Context` trait. The implementation of `Context` for `()` and `(A, C)` is updated correspondingly.

```
trait Context { type Length; ... }    impl Context for () { type Length = Z; ... }
impl<A: Slot, C: Context> Context for (A, C) { type Length = S<C::Length>; ... }
```

The $\mathsf{SILL_R}$ right rule $\mathrm{T}{\multimap}_{\mathsf{R}}$ is then encoded as follows:

```
fn receive_channel<A: Protocol, B: Protocol, C1: Context, C2: Context>(
  cont: impl FnOnce(C1::Length) -> PartialSession<C2, B>) ->
  PartialSession<C1, ReceiveChannel<A, B>> where C1: AppendContext<(A, ()), Appended=C2>
```

The function `receive_channel` is parameterized by a linear context `C1` implementing `AppendContext` to append the session type `A` to `C1`. The continuation argument `cont` is a closure that is given a context lens `C::Length`, and returns a `PartialSession` with `C2=C1::Appended` as its linear context. The function returns a `PartialSession` with linear context `C1`, offering session type `ReceiveChannel<A, B>`.

We note that in the type signature of `receive_channel`, the type `C1::Length` is not shown to have any `ContextLens` implementation. However when `C1::Length` is instantiated to the concrete types `Z`, `S<Z>`, etc in the continuation body, Rust will use the appropriate implementations of `ContextLens` so that they can be used to access the appended channel in the linear context.

The use of `receive_channel` is illustrated with the `hello_client` example below:

```
let hello_client = receive_channel(|a| {
  send_value_to(a, "Alice".to_string(), wait(a, terminate())) });
```

The `hello_client` program is inferred to have the Rust type `Session<ReceiveChannel<ReceiveValue<String, End>, End>>`. It is written to communicate with the `hello_provider` program defined earlier in Section 4.1. The interaction is achieved by having `hello_client` offering the session type `ReceiveChannel<ReceiveValue<String, End>, End>`. In its body, `hello_client` uses `receive_channel` to receive channel `a` of type `ReceiveValue<String, End>` from

`hello_provider`. The continuation closure is given an argument `a:Z`, denoting the context lens generated by `receive_channel` for accessing the received channel in the linear context. The context lens `a:Z` is then used for sending a string value, after which we `wait` for `hello_provider` to terminate. We note that the type `Z` of channel `a` (i.e. the channel position in the context) is automatically inferred by Rust and not exposed to the user.

4.3 Communication

At this point we have defined the necessary constructs to build and typecheck both `hello_provider` and `hello_client`, but the two are separate Ferrite programs that are yet to be linked with each other and executed.

$$\frac{\Gamma\,;\,\Delta_1 \vdash K_1 :: A \qquad \Gamma\,;\,\Delta_2, a : A \vdash K_2 :: B}{\Gamma\,;\,\Delta_1, \Delta_2 \vdash a \leftarrow \mathsf{cut}\; K_1\,;\; K_2 \;::\; B}\;(\text{T-CUT}) \qquad \frac{}{\Gamma\,;\, a : A \vdash \mathsf{forward}\, a \;::\; A}\;(\text{T-FWD})$$

In $\mathsf{SILL_R}$, rule T-CUT allows two session-typed programs to run in parallel, with the channel offered by K_1 added to the linear context of program K_2. Together with the forward rule T-FWD, we can use cut twice to run both `hello_provider` and `hello_client` in parallel, and have a third program that sends the channel offered by `hello_provider` to `hello_client`. The program `hello_main` would have the following pseudo code in $\mathsf{SILL_R}$:

$$\begin{aligned} hello_main : \epsilon \;=\;\; & f \leftarrow \mathsf{cut}\; hello_client; a \leftarrow \mathsf{cut}\; hello_provider; \\ & \mathsf{send_channel_to}\; f\; a; \mathsf{forward}\; f \end{aligned}$$

To implement `cut` in Ferrite, we need a way to split a linear context $\texttt{C} = \Delta_1, \Delta_2$ into two sub-contexts $\texttt{C1} = \Delta_1$ and $\texttt{C2} = \Delta_2$ so that they can be passed to the respective continuations. Moreover, since Ferrite programs use context lenses to access channels, the ordering of channels inside `C1` and `C2` must be preserved. We can preserve the ordering by replacing the corresponding slots with `Empty` during the splitting. Ferrite defines the `SplitContext` trait to implement the splitting as follows:

```
enum L {}      enum R {}
trait SplitContext<C: Context> { type Left: Context; type Right: Context; ... }
```

We first define two (uninhabited) marker types `L` and `R`. We then use type-level lists consisting of elements `L` and `R` to implement the `SplitContext` trait for a given linear context `C`. The `SplitContext` implementation contains the associated types `Left` and `Right`, representing the contexts `C1` and `C2` after splitting. As an example, the type `HList![L, R, L]` would implement `SplitContext<HList![A1, A2, A3]>` for any slot `A1`, `A2` and `A3`, with the associated type `Left` being `HList![A1, Empty, A3]` and `Right` being `HList![Empty, A2, Empty]`. We omit the implementation details of `SplitContext` for brevity. Using `SplitContext`, the function `cut` can be implemented as follows:

```
fn cut<XS, C: Context, C1: Context, C2: Context, C3: Context, A: Protocol, B: Protocol>
  ( cont1: PartialSession<C1, A>,
    cont2: impl FnOnce(C2::Length) -> PartialSession<C3, B> ) -> PartialSession<C, B>
where XS: SplitContext<C, Left=C1, Right=C2>, C2: AppendContext<HList![A], Appended=C3>
```

The function `cut` works by using the heterogeneous list `XS` that implements `SplitContext` to split a linear context `C` into `C1` and `C2`. To pass on the channel `A` that is offered by `cont1` to `cont2`, `cut` uses a similar technique to `receive_channel` to append the channel `A` to the end of `C2`, resulting in `C3`. Using `cut`, we can write `hello_main` in Ferrite as follows:

```
let hello_main: Session<End> = cut::<HList![]>(hello_client, |f| {
  cut::<HList![R]>(hello_provider, |a| { send_channel_to(f, a, forward(f)) }) });
```

Due to ambiguous instances for `SplitContext`, the type parameter `XS` has to be annotated explicitly for Rust to know in which context a channel should be placed. In the first use of `cut`, the context is empty, so we call `cut` with the empty list `HList![]`. We pass `hello_client` as the first continuation to run in parallel, and name the channel offered by `hello_client` as `f`. In the second use of `cut`, the linear context would be `HList![ReceiveValue<String, End>]`, with one channel `f`. We then have `cut` move `f` to the right side using `HList![R]`. On the left continuation, we have `hello_provider` run in parallel, and name the offered channel as `a`. In the right continuation, we use `send_channel_to` to send channel `a` to `f`. Finally, we forward the continuation of `f`, which now has type `End`.

Although `cut` provides the primitive way for Ferrite programs to communicate, its use can be cumbersome and requires a lot of boilerplate. For simplicity, we provide a specialized `apply_channel` construct that abstracts over the common usage pattern of `cut`. `apply_channel` takes a client program `f` offering session type `ReceiveChannel<A, B>` and a provider program `a` offering session type `A`, and sends `a` to `f` using `cut`. The use of `apply_channel` is akin to regular function application, making it more intuitive for programmers to use:

```
fn apply_channel<A: Protocol, B: Protocol>(
  f: Session<ReceiveChannel<A, B>>, a: Session<A>) -> Session<B>
```

4.4 Executing Ferrite Programs

To actually *execute* a Ferrite program, the program must offer some specific session types. In the simplest case, Ferrite provides the function `run_session` for running a top-level Ferrite program offering `End`, with an empty linear context:

```
async fn run_session(session: Session<End>) { ... }
```

Function `run_session` executes the session *asynchronously* using Rust's async/await infrastructure. Internally, `PartialSession<C, A>` implements the dynamic semantics of the Ferrite program, which is only accessible by public functions such as `run_session`. Ferrite currently uses the `tokio` [41] runtime for asynchronous execution, as well as the one shot channels from `tokio::sync::oneshot` to implement the low-level communication of Ferrite channels.

Since `run_session` accepts an argument of type `Session<End>`, this means that programmers must first use `cut` or `apply_channel` to fully link Ferrite programs with free channel variables, or Ferrite programs that offer session types other than `End` before they can be executed. This restriction ensures that all linear channels created in a Ferrite program are consumed. For example, the programs `hello_provider` and `hello_client` cannot be executed individually, but the program resulting from composing `hello_provider` with `hello_client` can be executed:

```
async fn main() { run_session(apply_channel(hello_client, hello_provider)).await; }
```

We omit the implementation details of the dynamics of Ferrite, which use low-level primitives such as Rust channels while carefully ensuring that the requirements and invariants of session types are satisfied. Interested readers can find more details in our companion technical report [7].

5 Recursive and Shared Session Types

Many real world applications, such as web services and instant messaging, implement protocols that are recursive in nature. As a result, it is essential for Ferrite to support recursive session types. In this section, we report on Rust's limited support for recursive types and how Ferrite addresses this limitation. We then discuss our encoding of *shared*, recursive session types.

5.1 Recursive Session Types

Consider a simple example of a counter session type, which sends an infinite stream of integer values, incrementing each by one. To write a Ferrite program that offers such a session type, we may attempt to define the counter session type as `type Counter = SendValue<u64, Counter>`. If we try to use such a type definition, the compiler will emit the error “cycle detected when processing `Counter`”. The issue with the definition is that it is a directly self-referential type alias, which is not supported in Rust. Rust imposes various restrictions on the legal forms of recursive types to ensure that the memory layout of data is known at compile-time.

Type-Level Fixed Points

To address this limitation, we implement type-level fixed points using *defunctionalization* [36, 46]. This is done by introducing a `RecApp` trait that is implemented by defunctionalized types that can be “applied” with a type parameter:

```
trait RecApp<X> { type Applied; }   type AppRec<F, X> = <F as RecApp<X>>::Applied;
struct Rec<F: RecApp<Rec<F>>> { unfold: Box<AppRec<F, Rec<F>>> }
```

The `RecApp` trait is parameterized by a type `X`, which serves as the type argument to be applied to. This makes it possible for a Rust type `F` that implements `RecApp` to act as if it has the higher-kinded type Type → Type, and be “applied” to type `X`. We define a type alias `AppRec<F, X>` to refer to the associated type `Applied` resulting from “applying” `F` to `X` via `RecApp`. Using `RecApp`, we can now define a type-level recursor `Rec` as a struct parameterized by a type `F` that implements `RecApp<Rec<F>>`. The body of `Rec` contains a boxed value `Box<AppRec<F, RecApp<Rec<F>>>>` to make it have a fixed size in Rust.

Ferrite implements `RecApp` for all `Protocol` types, with the type `Z` used to denote the recursion point. With that, the example `Counter` type would be defined as `type Counter = Rec<SendValue<u64, Z>>`. The type `Rec<SendValue<T, Z>>` is unfolded into `SendValue<T, Rec< SendValue<T, Z>>>` through generic implementations of `RecApp` for `SendValue` and `Z`:

```
impl<X> RecApp<X> for Z { type Applied = X; }
impl<X, T, A: RecApp<X>> RecApp<X> for SendValue<T, A> {
  type Applied = SendValue<T, AppRec<A, X>; }
```

Inside `RecApp`, `Z` simply replaces itself with the type argument `X`. `SendValue<T, A>` delegates the type application of `X` to `A`, provided that the session type `A` also implements `RecApp` for `X`.

The session type `Counter` is iso-recursive, as the rolled type `Rec<SendValue<u64, Z>>` and the folded type `SendValue<u64, Rec<SendValue<u64, Z>>` are considered distinct types in Rust. As a result, Ferrite provides the constructs `fix_session` and `unfix_session` for converting between the rolled and unfolded versions of a recursive session type.

Nested Recursive Session Types

The use of `RecApp` is akin to emulating the higher-kinded type (HKT) Type → Type in Rust. As of this writing, HKTs are only available in the nightly (unstable) version of Rust through *generic associated types*. However, even with support for HKTs, our defunctionalization-based approach via `RecApp` allows us to generalize to *nested* recursive types.

To account for a recursive type with multiple recursion points, we introduce a *recursion context* `R` as a type-level list of elements (c.f. the linear context of Section 4.2). The type-level natural numbers `Z`, `S<Z>`, etc. are now used as de Bruijn indices to unfold to the elements in the recursion context. The type-level fixed point combinator `Rec` is redefined as `RecX`, containing the recursion context:

```
struct RecX<R, F: RecApp<(RecX<R, F>, R)>> { unfix: Box<AppRec<F, (RecX<R, F>, R)>> }
type Rec<F> = RecX<(), F>;
impl<R, F: RecApp<(RecX<R, F>, R)>> RecApp<R> for RecX<(), F>  {
    type Applied = RecX<R, F>; }
```

A recursive session type is defined starting with an empty recursion context. Since nested recursive session types allow a `RecX` to be embedded inside another `RecX`, we have `RecX` also implement `RecApp`, provided it has an empty recursion context. When unfolded from another recursion context `R`, `RecX` simply saves `R` as its own recursion context and does not unfold further in `F`. The inner type `F` is only unfolded once with the full recursion context after all surrounding `RecX` types are unfolded.

The recursive marker `Z` is modified to unfold to the first element of the recursion context. We then implement `S<N>` to unfold to the (N+1)-th position in the recursion context:

```
impl<A, R> RecApp<(A, R)> for Z { type Applied = A; }
impl<A, R, N: RecApp<R>> RecApp<(A, R)> for S<N> { type Applied = N::Applied; }
```

5.2 Shared Session Types

In the previous section we explored a recursive session type `Counter`, which is defined using `Rec` and `Z`. Since `Counter` is defined as a linear session type, it cannot be shared among multiple clients. Shared communication, however, is essential to implement many practical applications. For instance, we may want to implement a simple counter web-service, to send a unique count for each request. To support such shared communication, we introduce *shared session types* in Ferrite, enabling *safe* shared communication in the presence multiple clients.

Shared Session Types in Ferrite

As introduced in Section 2, the SILL_S (and SILL_R) notion of shared session types is recursive in nature, as a shared session type must offer the same linear critical section to all clients that acquire a shared resource. For instance, a shared version of the `Counter` type in SILL_R is:

$$\mathsf{SharedCounter} = \uparrow^{\mathsf{S}}_{\mathsf{L}}\mathsf{Int} \triangleleft \downarrow^{\mathsf{S}}_{\mathsf{L}}\mathsf{SharedCounter}$$

The linear portion of $\mathsf{SharedCounter}$ in between $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ (acquire) and $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ (release) amounts to a critical section. When a $\mathsf{SharedCounter}$ is *acquired*, it offers a linear session type $\mathsf{Int} \triangleleft \downarrow^{\mathsf{S}}_{\mathsf{L}}\mathsf{SharedCounter}$, willing to send an integer value, after which it must be *released* to become available again as a $\mathsf{SharedCounter}$ to the next client.

The recursive aspect of shared session types in SILL_R means that we can reuse the implementation technique that we use for recursive session types. The type `SharedCounter` can be defined in Ferrite as follows:

```
type SharedCounter = LinearToShared<SendValue<u64, Release>>;
```

Compared to linear recursive session types, the main difference is that instead of using `Rec`, a shared session type is defined using the `LinearToShared` construct. This corresponds to $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ in SILL_R, with the inner type `SendValue<u64, Release>` corresponding to the linear portion of the shared session type. At the point of recursion, the type `Release` is used in place of $\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathsf{SharedCounter}$. As a result, the type `LinearToShared<SendValue<u64, Release>>` is unfolded into `SendValue<u64, SharedToLinear<LinearToShared<SendValue<u64, Release>>>>` after being acquired. Type unfolding is implemented as follows:

```
trait SharedRecApp<X> { type Applied; }   trait SharedProtocol { ... }
struct SharedToLinear<F> { ... }          struct LinearToShared<F> { ... }
impl<F> Protocol for SharedToLinear<LinearToShared<F>>
```

```
  where F: SharedRecApp<SharedToLinear<LinearToShared<F>>> { ... }
impl<F> SharedProtocol for LinearToShared<F>
  where F: SharedRecApp<SharedToLinear<LinearToShared<F>>> { ... }
```

The struct `LinearToShared` is parameterized by a linear session type `F` that implements the trait `SharedRecApp<SharedToLinear<LinearToShared<F>>>`. It uses the `SharedRecApp` trait instead of the `RecApp` trait to ensure that the session type is *strictly equi-synchronizing* [3], requiring an acquired session to be released to the same type at which it was previously acquired. Ferrite enforces this requirement by omitting an implementation of `SharedRecApp` for `End`, ruling out invalid shared session types such as `LinearToShared<SendValue<u64, End>>`. We note that the type argument to `F`'s `SharedRecApp` is another struct `SharedToLinear`, which corresponds to $\downarrow^{S}_{L}$ in $\mathsf{SILL_R}$. A `SharedProtocol` trait is also defined to identify shared session types, i.e. `LinearToShared`.

Once a shared process is started, a shared channel is created to allow multiple clients to access the shared process through the use of shared channel:

```
struct SharedChannel<S: SharedProtocol>{...}   impl<S> Clone for SharedChannel<S>{...}
```

The code above shows the definition of the `SharedChannel` struct. Unlike linear channels, shared channels follow structural typing, i.e. they can be weakened or contracted. This means that we can delegate the handling of shared channels to Rust, given that `SharedChannel` implements Rust's `Clone` trait to allow contraction. Whereas $\mathsf{SILL_S}$ provides explicit constructs for sending and receiving shared channels, Ferrite's shared channels can be sent as regular Rust values using `Send/ReceiveValue`.

On the client side, a `SharedChannel` serves as an endpoint for interacting with a shared process running in parallel. To start the execution of such a shared process, a corresponding Ferrite program has to be defined and executed. Similar to `PartialSession`, we define `SharedSession` as shown below to represent such a shared Ferrite program.

```
struct SharedSession<S: SharedProtocol> { ... }
fn run_shared_session<S: SharedProtocol>(session: SharedSession<S>) -> SharedChannel<S>
```

Just as `PartialSession` encodes linear Ferrite programs without executing them, `SharedSession` encodes shared Ferrite programs without executing them. Since `SharedSession` does not implement the `Clone` trait, the shared Ferrite program is itself affine and cannot be shared. To enable sharing, the shared Ferrite program must first be executed with `run_shared_session`. The function `run_shared_session` takes a shared Ferrite program of type `SharedSession<S>` and starts it in the background as a shared process. Then, in parallel, the shared channel of type `SharedChannel<S>` is returned to the caller, which can then be sent to multiple clients for access to the shared process.

Below we demonstrate how a shared session can be defined and used by multiple clients:

```
type SharedCounter = LinearToShared<SendValue<u64, Release>>;
fn counter_producer(current_count: u64) -> SharedSession<SharedCounter> {
  accept_shared_session(async move {
    send_value(current_count, detach_shared_session(
        counter_producer(current_count + 1))) }) }
fn counter_client(counter: SharedChannel<SharedCounter>) -> Session<End> {
  acquire_shared_session(counter, move | chan | {
    receive_value_from(chan, move | count | { println!("received count: {}", count);
      release_shared_session(chan, terminate()) }) }) }
```

The recursive function `counter_producer` creates a `SharedSession` program that, when executed, offers a shared channel of session type `SharedCounter`. On the provider side, a shared session is defined using the `accept_shared_session` construct, with a continuation given as an async thunk that is executed when a client acquires the shared session and enters

the linear critical section (of type `SendValue<u64, SharedToLinear<SharedCounter>>`). Inside the closure, the producer uses `send_value` to send the current count to the client and then uses `detach_shared_session` to exit the linear critical section. The construct `detach_shared_session` offers the linear session type `SharedToLinear<SharedCounter>` and expects a continuation that offers the shared session type `SharedCounter` to serve the next client. We generate the continuation by recursively calling the `counter_producer` function.

The `counter_client` function takes a shared channel of session type `SharedCounter` and returns a session type program that acquires the shared channel and prints the received count value to the terminal. A linear Ferrite program can acquire a shared session using the `acquire_shared_session` construct, which accepts a `SharedChannel` object and adds the acquired linear channel to the linear context. In this case, the continuation closure is given the context lens `Z`, which provides access to the linear channel of session type `SendValue <u64, SharedToLinear<SharedCounter>>` in the first slot of the linear context. It then uses `receive_value_from` to receive the value sent by the shared provider and then prints the value. On the client side, the linear session of type `SharedToLinear<SharedCounter>` must be released using the `release_shared_session` construct. After releasing the shared session, other clients will then be able to acquire the shared session.

```
async fn main () {
  let counter1: SharedChannel<SharedCounter> = run_shared_session(counter_producer(0));
  let counter2 = counter1.clone();
  let child1 = task::spawn(async move { run_session(counter_client(counter1)).await; });
  let child2 = task::spawn(async move { run_session(counter_client(counter2)).await; });
  join!(child1, child2).await; }
```

To illustrate a use of `SharedCounter`, we have a `main` function that initializes a shared producer with an initial value of 0 and then runs the shared provider using the `run_shared_session` construct. The returned `SharedChannel` is then cloned, making the shared counter accessible via aliases `counter1` and `counter2`. It then uses `task::spawn` to spawn two async tasks that run `counter_client` twice. A key observation is that multiple Ferrite programs that are executed independently can access *the same* shared producer through a reference to the shared channel.

A follow up example of `SharedQueue`, which demonstrates the Ferrite implementation of the $\mathsf{SILL_S}$ shared queue example in Section 2 is available in our companion technical report [7].

6 Choice

Session types support *internal* and *external* choice, leaving the choice among several options to the provider or the client, resp. (see Table 2). When restricted to binary choice, the implementation is relatively straightforward, as shown below by the two right rules for internal choice in $\mathsf{SILL_R}$. The offer_left and offer_right constructs allow a provider to offer an internal choice $A \oplus B$ by offering either A or B, resp.

$$\frac{\Gamma\,;\,\Delta\,\vdash\,K :: A}{\Gamma\,;\,\Delta\,\vdash\,\mathsf{offer_left};\,K :: A \oplus B}\;(\mathrm{T}{\oplus}2\mathrm{L_R}) \qquad \frac{\Gamma\,;\,\Delta\,\vdash\,K :: B}{\Gamma\,;\,\Delta\,\vdash\,\mathsf{offer_right};\,K :: A \oplus B}\;(\mathrm{T}{\oplus}2\mathrm{R_R})$$

It is straightforward to implement the two versions of the right rules by writing the two respective functions `offer_left` and `offer_right`:

```
fn offer_left<C: Context, A: Protocol, B: Protocol>
  ( cont: PartialSession<C, A> ) -> PartialSession<C, InternalChoice2<A, B>>
fn offer_right < C: Context, A: Protocol, B: Protocol >
  ( cont: PartialSession<C, B> ) -> PartialSession<C, InternalChoice2<A, B>>
```

However, this approach does not scale if we want to generalize choice beyond two options. To support n-ary choice, the functions would have to be explicitly reimplemented N times. Instead, we implement a single `offer_case` function which allows selection from n-ary branches.

In Section 4.2, we explored heterogeneous lists to encode the linear context, i.e. *products* of session types of arbitrary lengths. We then implemented context *lenses* to access and update individual channels in the linear context. Observing that n-ary choices can be encoded as *sums* of session types, we now use *prisms* to implement the selection of an arbitrary-length branch. Ferrite also supports an n-ary choice type `InternalChoice<HList![...]>`, with `InternalChoice<HList![A, B]>` being the special case of a binary choice. To select a branch out of the heterogeneous list, we define the `Prism` trait as follows:

```
trait Prism<Row> {type Elem; ...}  impl<A, R> Prism<(A, R)> for Z {type Elem = A; ... };
impl<N, A, R> Prism<(A, R)> for S<N> where N: Prism<R> { type Elem = N::Elem; ... }
```

The `Prism` trait is parameterized over a row type `Row=HList![...]`, with the associated type `Elem` being the element type that has been selected from the list by the prism. We then inductively implement `Prism` using type-level natural numbers, with the number `N` used for selecting the N-th element of the heterogeneous list. The definition of `Prism` is similar to `ContextLens`, with the main difference being that we only need `Prism` to support extraction and injections operations on the sum types that are derived from the heterogeneous list. Using `Prism`, a generalized `offer_case` function is implemented as follows:

```
fn offer_case<C: Context, A: Protocol, Row, N: Prism<Row, Elem=A>>
  (n: N, cont: PartialSession<C, A>) -> PartialSession<C, InternalChoice<Row>>
```

The function accepts a natural number `N` as the first parameter, which acts as the *prism* for selecting a session type A_N out of the row type `Row=HList![..., A_N, ...]`. Through the associated type `A=N::Elem`, `offer_case` forces the programmer to provide a continuation that offers the chosen session type `A`.

While `offer_case` is a step in the right direction, it only allows the selection of a specific choice, but not the provision of *all* possible choices. The latter, however, is necessary to encode the $\mathsf{SILL_R}$ left rule of internal choice and right rule of external choice. To illustrate the problem, let's consider the right rule of a binary external choice, $\mathrm{T}\&2_{\mathsf{R}}$:

$$\frac{\Gamma\,;\,\Delta\,\vdash\,K_l :: A \qquad \Gamma\,;\,\Delta\,\vdash\,K_r :: B}{\Gamma\,;\,\Delta\,\vdash\,\mathsf{offer_choice_2}\;K_l\;K_r :: A\&B}\;(\mathrm{T}\&2_{\mathsf{R}})$$

The `offer_choice_2` construct has two possible continuations K_l and K_r, with only one of them being executed, depending on the selection by the client. In a naive implementation, we can define the construct to accept two continuations as follows:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont_left: PartialSession<C, A>, cont_right: PartialSession<C, B> )
  -> PartialSession<C, ExternalChoice2<A, B>>
```

While the above implementation works in most languages, it is not adequate in Rust. Since Rust's type system is *affine*, variables can only be captured by one of the continuation closures, but not both. As far as the compiler is aware, both closures can potentially be called, and we cannot state that one of the branches is guaranteed to never run.

In order for `offer_choice_2` to work in Rust's affine typing, it has to accept only one continuation closure and have it return either `PartialSession<C, A>` or `PartialSession<C, B>`, depending on the client's selection. It is not as straightforward to express such behavior as a valid type in a language like Rust. If Rust supported dependent types, `offer_choice_2` could be implemented along the following lines:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont: impl FnOnce(first: bool) -> if first { PartialSession<C, A> }
     else { PartialSession<C, B> } ) -> PartialSession<C, ExternalChoice2<A, B>>
```

That is, the return type of the `cont` closure depends on the whether the *value* of the `first` argument is true or false. However, since Rust does not support dependent types, we emulate a dependent sum in a non-dependent language, using a CPS transformation:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont: impl FnOnce(InjectSum2<C, A, B>) -> ContSum2<C, A, B> )
    -> PartialSession<C, ExternalChoice2<A, B>>
```

The function `offer_choice_2` accepts a continuation function `cont` that is given a value of type `InjectSum2<C, A, B>` and returns a value of type `ContSum2<C, A, B>`. We will now look at the definitions of `ContSum2` and `InjectSum2`. First, we observe that the different return types for the two branches can be unified with a type `ContSum2`:

```
struct ContSum2<C: Context, A: Protocol, B: Protocol> { ... }
async fn run_cont_sum<C: Context, A: Protocol, B: Protocol>(cont: ContSum2<C, A, B>)
```

The type `ContSum2` contains the necessary data for executing either a `PartialSession<C, A>` or a `PartialSession<C, B>`, together with the runtime data for the linear context `C`. For brevity, the implementation details of `ContSum2` are omitted, with the private function `run_cont_sum` provided as an abstraction for Ferrite to execute the continuation.

We then define `InjectSum2` as a sum of boxed closures that would construct a `ContSum2` from either a `PartialSession<C, A>` or a `PartialSession<C, B>`:

```
enum InjectSum2<C, A, B> {
  InjectLeft(Box<dyn FnOnce(PartialSession<C, A>) -> ContSum2<C, A, B>>),
  InjectRight(Box<dyn FnOnce(PartialSession<C, B>) -> ContSum2<C, A, B>>) }
```

When the `cont` passed to `offer_choice_2` is given a value of type `InjectSum2<C, A, B>`, it has to branch on it and match on whether the `InjectLeft` or `InjectRight` constructors are used. Since the return type of `cont` is `ContSum2<C, A, B>` and the constructor for `ContSum2` is private, there is no other way for `cont` to construct the return value other than to call either `InjectLeft` or `InjectRight` with the appropriate continuation.

The use of `InjectSum2` prevents the programmer from providing the wrong branch in the continuation by keeping the constructor private. However, a private constructor alone cannot prevent two uses of `InjectSum2` to be *deliberately* interchanged, causing a protocol violation. To fully ensure that there is no way for the user to provide a `ContSum2` from elsewhere, we instead use a technique from GhostCell [47] that uses *higher-ranked trait bounds* (HTRB) to mark a phantom invariant lifetime on both `InjectSum2` and `ContSum2`:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont: for <'r> impl FnOnce(InjectSum2<'r, C, A, B>) -> ContSum2<'r, C, A, B> )
    -> PartialSession<C, ExternalChoice2<A, B>>
```

The use of HRTB ensures that each call of `offer_choice_2` would generate a unique lifetime `'r` for the continuation. Using that, Ferrite can ensure that a value of type `InjectSum2<'r1, C, A, B>` cannot be used to construct the return value of type `ContSum2<'r2, C, A, B>`, if the lifetimes `<'r1>` and `<'r2>` are different. An example use of `offer_choice_2` is as follows:

```
let choice_provider: Session<ExternalChoice2<SendValue<u64, End>, End>>
  = offer_choice_2(|b| { match b { InjectLeft(ret) => ret(send_value(42, terminate())),
                                   InjectRight(ret) => ret(terminate()) } });
```

To free the programmer from writing such boilerplate, Ferrite also provides macros that translates into the underlying pattern matching syntax. The macros allow the same example to be written as follows:

```
1 enum CanvasMsg { Canvas2d(Canvas2dMsg, CanvasId), Close(CanvasId), ... }
2 enum Canvas2dMsg { LineTo(Point2D), GetTransform(Sender<Transform2D>),
3   IsPointInPath(f64, f64, FillRule, IpcSender<bool>), ... }
4 enum ConstellationCanvasMsg { Create { id_sender: Sender<CanvasId>, size: Size2D } }
5 struct CanvasPaintThread { canvases: HashMap<CanvasId, CanvasData>, ... }
6 impl CanvasPaintThread { ...
7   fn start() -> (Sender<ConstellationCanvasMsg>, Sender<CanvasMsg>) {
8     let (msg_sender, msg_receiver) = channel();  let (create_sender, create_receiver) = channel();
9     thread::spawn(move || { loop { select! {
10         recv(canvas_msg_receiver) -> { ...
11           CanvasMsg::Canvas2d(message, canvas_id) => { ...
12             Canvas2dMsg::LineTo(point) => self.canvas(canvas_id).move_to(point),
13             Canvas2dMsg::GetTransform(sender) =>
14               sender.send(self.canvas(canvas_id).get_transform()).unwrap(), ... }
15           CanvasMsg::Close(canvas_id) => canvas_paint_thread.canvases.remove(&canvas_id) }
16         recv(create_receiver) -> { ... ConstellationCanvasMsg::Create { id_sender, size } => {
17             let canvas_id = ...; self.canvases.insert(canvas_id, CanvasData::new(size, ...));
18             id_sender.send(canvas_id); } } } } });
19     (create_sender, msg_sender) }
20   fn canvas(&mut self, canvas_id: CanvasId) -> &mut CanvasData {
21     self.canvases.get_mut(&canvas_id).expect("Bogus canvas id") } }
```

Figure 1 Message-passing concurrency in Servo's canvas component (simplified for illustration purposes).

```
define_choice!{ CustomChoice; Left: SendValue<u64, End>, Right: End }
let choice_provider: Session<ExternalChoice<CustomChoice>> = offer_choice! {
  Left => send_value(42, terminate()), Right => terminate() };
```

The `define_choice!` macro allows defining named n-ary branches of choice. The `offer_choice!` macro allows the choice provider to branch without the boilerplate used in the earlier example. The generalization from binary to n-ary choice is omitted for conciseness. The details can be found in our companion technical report [7].

7 Evaluation

The Ferrite library is more than just a research prototype. It is designed for practical use in real world applications. To evaluate the design and implementation of Ferrite, we re-implemented the communication layer of the canvas component of Servo [29] entirely in Ferrite. Servo is an under development browser engine that uses message-passing for heavy task parallelization. Canvas provides 2D graphic rendering, allowing clients to create new canvases and perform operations on a canvas such as moving the cursor and drawing shapes.

The canvas component is a good target for evaluation as it is sufficiently complex and also very demanding in terms of performance. Canvas is commonly used for animations in web applications. For an animation to look smooth, a canvas must render at least 24 frames per second, with potentially thousands of operations to be executed per frame.

The changes we made are fairly minimal, consisting of roughly 750 lines of additions and 620 lines of deletions, out of roughly 300,000 lines of Rust code in Servo. The sources of our implementation are provided as an artifact. To differentiate the two versions of code snippets, we use blue for the original code, and green for the code using Ferrite.

7.1 Servo Canvas Component

Figure 1 provides a sketch of the main communication paths in Servo's canvas component [30]. The canvas component is implemented by the `CanvasPaintThread`, whose function `start` contains the main communication loop running in a separate thread (lines 9–18). This loop processes client requests received along `canvas_msg_receiver` and `create_receiver`, which are the receiving endpoints of the channels created prior to spawning the loop (lines 8–8). The channels are typed with the enumerations `ConstellationCanvasMsg` and `CanvasMsg`, defining

messages for creating and terminating the canvas component and for executing operations on an individual canvas, resp. When a client sends a message that expects a response from the recipient, such as `GetTransform` and `IsPointInPath` (lines 2–3), it sends a channel along with the message to be used by the recipient to send back the result. Canvases are identified by an id, which is generated upon canvas creation (line 17) and stored in the thread's `canvases` hash map (line 5). If a client requests an invalid id, for example after prior termination and removal of the canvas (line 15), the failed assertion `expect("Bogus canvas id")` (line 21) will result in a `panic!`, causing the canvas component to crash and subsequent calls to fail.

The code in Figure 1 uses a clever combination of enumerations to type channels and ownership to rule out races on the data sent along channels. Nonetheless, Rust's type system is not expressive enough to enforce the intended *protocol* of message exchange and existence of a communication partner. The latter is a consequence of Rust's type system being *affine*, which permits "dropping of a resource". The dropping or premature closure of a channel, however, can result in a proliferation of `panic!` and thus cause an entire application to crash. In fact, while refactoring Servo to use Ferrite, we were able to uncover a protocol violation in Servo, caused by one of the nested match arms of the provider doing an early return before sending back any result to the client.

7.2 Canvas Protocol in Ferrite

In the original canvas component, the provider `CanvasPaintThread` accepts messages of type `CanvasMsg`, made up of a combination of smaller sub-message types such as `Canvas2dMsg`. We note that the majority of the sub-message types have the following trivial form:

```
enum CanvasMsg { Canvas2d(Canvas2dMsg, CanvasId), Close(CanvasId), ... }
enum Canvas2dMsg { BeginPath, ClosePath, Fill(FillOrStrokeStyle), ... }
```

The trivial sub-message types such as `BeginPath`, `Fill`, and `LineTo` do not require a response from the provider, so the client can simply fire them and proceed. Although we can offer all sub-message types as separate branches in an external choice, it is more efficient to keep trivial sub-messages in a single enum. In our implementation, we define `CanvasMessage` to have similar sub-messages as `Canvas2dMsg`, with non-trivial messages such as `IsPointInPath` moved to separate branches.

```
enum CanvasMessage { BeginPath, ClosePath, Fill(FillOrStrokeStyle), ... }
define_choice! { CanvasOps; Message: ReceiveValue<CanvasMessage, Release>, ... }
type Canvas = LinearToShared<ExternalChoice<CanvasOps>>;
```

We use the `define_choice!` macro described in Section 6 to define an n-ary choice `CanvasOps`. The first branch of `CanvasOps` is labelled `Message`, and the only action is for the provider to receive a `CanvasMessage`. The choices are offered as an external choice, and the session type `CanvasProtocol` is defined as a shared protocol that offers the choices in the critical section.

The original design of the `CanvasPaintThread` would be sufficient if the only messages being sent were trivial messages. However, `Canvas2dMsg` also contains non-trivial sub-messages, such as `GetImageData` and `IsPointInPath`, demanding a response from the provider:

```
enum Canvas2dMsg { ..., GetImageData(Rect<u64>, Size2D<u64>, IpcBytesSender),
  IsPointInPath(f64, f64, FillRule, IpcSender<bool>), ... }
```

To obtain the result from the original canvas, clients must create a new inter-process communication (IPC) channel and bundle the channel's sender endpoint with the message. In our implementation, we define separate branches in `CanvasOps` to handle non-trivial cases:

```
define_choice! { CanvasOps; Message: ReceiveValue<CanvasMessage, Release>,
  GetImageData: ReceiveValue<(Rect<u64>, Size2D<u64>), SendValue<ByteBuf, Release>>,
  IsPointInPath: ReceiveValue<(f64, f64, FillRule), SendValue<bool, Release>>, ... }
```

Table 4 MotionMark Benchmark scores in fps (higher is better).

Benchmark Name	Servo	Servo/Ferrite	Firefox	Chrome
Arcs	12.21 ± 6.75%	11.83 ± 11.49%	52.61 ± 32.88%	46.00 ± 9.00%
Paths	43.76 ± 10.66%	40.98 ± 18.94%	55.59 ± 28.80%	59.50 ± 14.90%
Lines	7.48 ± 7.06%	11.47 ± 12.74%	14.35 ± 6.65%	32.43 ± 6.48%
Bouncing clipped rects	18.43 ± 7.06%	18.23 ± 11.00%	34.82 ± 7.76%	58.07 ± 19.85%
Bouncing gradient circles	8.02 ± 7.74%	7.72 ± 12.63%	58.79 ± 21.03%	59.77 ± 10.07%
Bouncing PNG images	7.97 ± 5.91%	6.31 ± 10.26%	24.61 ± 6.35%	59.94 ± 13.04%
Stroke shapes	10.60 ± 3.95%	10.35 ± 10.96%	51.21 ± 11.25%	59.38 ± 16.87%
Put/get image data	60.01 ± 3.81%	32.08 ± 10.83%	59.66 ± 20.16%	60.00 ± 5.00%

The original `GetImageData` accepts an `IpcBytesSender`, which sends raw bytes back to the client. In Ferrite, we translate the use of `IpcBytesSender` to the type `SendValue<ByteBuf, Z>`, which sends the raw bytes wrapped in a `ByteBuf` type.

Aside from the `Canvas` protocol, we also redesign the use of `ConstellationCanvasMsg` into its own shared protocol, `ConstellationCanvas`:

```
type ConstellationCanvas = LinearToShared<ReceiveValue<Size2D,
    SendValue<SharedChannel<Canvas>, Release>>>;
```

To create a new canvas, a client first acquires the shared channel of type `SharedChannel<ConstellationCanvas>`. Afterwards, the client sends the `Size2D` parameter to specify the canvas size. The constellation canvas provider then spawns a new canvas shared process through `run_shared_session` and sends back the shared channel of type `SharedChannel<Canvas>` as a value. Finally, the session is released, allowing other clients to acquire the shared provider.

7.3 Performance Evaluation

To evaluate the performance of the canvas component, we use the MotionMark benchmark suite [45]. MotionMark is a web benchmark that focuses on graphics performance of web browsers. It contains benchmarks for various web components, including canvas, CSS, and SVG. As MotionMark does not yet support Servo, we modified the benchmark code to make it work in the absence of features that are not implemented in Servo (details on the benchmarks can be found in the companion artifact).

For the purpose of this evaluation, we focused on benchmarks that target the canvas component and skipped benchmarks that fail in Servo due to missing features. We ran each benchmark in a fixed 1600x800 resolution for 30 seconds, on a Core i7 Linux desktop machine. We ran the benchmarks against the original Servo, modified Servo with Ferrite canvas (Servo/Ferrite), Firefox (v98), and Chrome (v99). Our performance scores are measured in the fixed mode version of MotionMark, which measures frames per second (fps) performance of executing the same set of canvas operations per frame.

The benchmark results are shown in Table 4, with the performance scores in fps (higher fps is better). It is worth noting that a benchmark can achieve at most 60 fps. Our goal in this benchmark is to keep the scores of Servo/Ferrite close to those of Servo, *not* to achieve better performance than the original. This is shown to be the case in most of the benchmarks.

The only benchmark with a large difference between Servo and Servo/Ferrite is *Put/get image data*, with Ferrite performing 2x worse. This is because in Servo/Ferrite, we use `ByteBuf` to transfer the images as raw bytes within the same shared channel. Servo uses a specialized structure `IpcBytesSender` for transferring raw bytes in parallel to other messages. As a result, communication in Servo/Ferrite is congested during the transfer of the image data, while the original Servo can process new messages in parallel with the image transmission.

We also observe that there are significant performance differences in the scores between Servo and those in Firefox and Chrome, indicating that there exist performance bottlenecks in Servo unrelated to communication protocols.

8 Related and Future Work

Session type embeddings exist for various languages, including Haskell [34, 20, 27, 31], OCaml [32, 19], Java [18, 17], and Scala [39]. Functional languages like ML, OCaml, and Haskell, in particular, are ideal host languages for creating EDSLs thanks to their advanced features (e.g. type classes, type families, higher-rank and higher-kinded types and GADTs). [34] first demonstrated the feasibility of embedding session types in Haskell, with refinements done in later works [20, 27, 31]. Similar embeddings have also been contributed in the context of OCaml by `FuSe` [32] and `session-ocaml` [19].

Aside from Ferrite, there are other implementations of session types in Rust, including `session_types` [21], `sesh` [25], and `rumpsteak` [8, 9]. `session_types` were the first implementation to make use of affinity to provide a session type library in Rust. `sesh` emphasizes this aspect by embedding the affine session type system Exceptional GV [12] in Rust. Both `session_types` and `sesh` adopt a classical perspective, requiring the endpoints of a channel to be typed with dual types. `rumpsteak` develops an embedding of multiparty session types by generating Rust types derived from multiparty session types defined in Scribble [48].

Due to their reliance on Rust's affine type system, neither `session_types` nor `sesh` prevent a channel endpoint from being dropped prematurely, relegating the handling of such errors to the runtime. `rumpsteak` uses some type-level techniques similar to Ferrite to enforce a channel's linear usage in the continuation passed to the `try_session` function. This ensures that a linear channel in `rumpsteak` is always fully consumed, if it is ever consumed. However, prior to the call to `try_session`, the linear channel exists as an affine value, which may be dropped by without being consumed at all, resulting in a deadlock. Ferrite enforces linearity *at all levels*, including safe linking of multiple linear processes using `cut`.

In terms of concurrency, `session_types`, `sesh`, and `rumpsteak` all require the programmer to manually manage concurrency, either by spawning threads or async tasks. This introduces potential failure when the code fails follow the requirement to spawn all processes. On the other hand, the simplicity of such a model allows relatively few threads or async tasks to be spawned, thereby allowing the underlying runtime to execute the processes more efficiently. In comparison, Ferrite offers fully managed concurrency, without the programmer having to worry about how to spawn the processes and execute them in parallel.

In terms of performance, the downside of Ferrite's concurrency approach is that it aggressively spawns new async tasks in each use of `cut`. Although async tasks in Rust are much more lightweight than OS threads, there is still a significant overhead in spawning and managing many async tasks, especially in micro-benchmarks. As a result, Ferrite tends to perform slower than alternative Rust implementations in settings where only a fixed small number of processes need to be spawned. Nevertheless, it is worth noting that the async ecosystem in Rust is still relatively immature, with many potential improvements to be made. In practice, the overhead of the async runtime may also be negligible when compared to the core application logic. In such cases, Ferrite would also allow applications to scale more easily by allowing many more processes to be spawned and managed concurrently without requiring additional effort from the programmer.

In terms of DSL design, Ferrite is closely related to the embeddings in OCaml and Haskell, as it fully enforces a linear treatment of channels and thus *statically* rules out any panics arising from dropping a channel prematurely. However, Ferrite leverages Rust's affine type

system, which naturally extends to support linear types as compared to the structural type systems of OCaml or Haskell. As a result, Ferrite programs can reuse any existing Rust code without sacrificing the benefit of affine types. This is generally not the case with substructural EDSLs, which often require rewriting of libraries (e.g. LinearHaskell's `linear-base`).

Ferrite also differs from other libraries in that it adopts intuitionistic typing [4], allowing the typing of a channel rather than its two endpoints via type duality. While the use of dual types is convenient for simple types like `ReceiveValue<String, End>`, the mental overhead of computing the dual type becomes higher when higher-order channels are involved. For example, when implementing a process with type `ReceiveChannel<ReceiveValue<String, End>>`, the programmer would have to keep in mind that the received channel would have its session type flipped and become `SendValue<String, End>`. From an ergonomics point of view, we believe that intuitionistic session types provide a more familiar model of programming.

On the use of profunctor optics, our work is the first to connect n-ary choice to prisms, while prior work by `session-ocaml` [20] has only established the connection between lenses, the dual of prisms, and linear contexts. `FuSe` [32] and `session-ocaml` [19] have previously explored the use of n-ary (generalized) choice through extensible variants available only in OCaml. Our work demonstrates that it is possible to encode extensible variants, and thus n-ary choice, as type-level constructs using features available in Rust.

A major difference in terms of implementation is that Ferrite uses a continuation-passing style, whereas Haskell and OCaml embeddings commonly use (indexed) monads and do-notation. This technical difference amounts to a key conceptual one: a direct correspondence between the Rust programs generated from Ferrite constructs and the $\mathsf{SILL_R}$ typing derivation. As a result, the generated Rust code can be viewed as carrying the proof of protocol adherence.

The embeddings of `ESJ` [17] and `lchannels` [39] also adopt a continuation-passing style, but do not faithfully embed typing derivations (i.e. they do not statically enforce linearity). They follow an encoding of session types using linear types [10] first proposed by Kobayashi [24] in the setting of π-calculus. While session types are generally less powerful than the approaches of Kobayashi et al., they provide a useful compromise between expressiveness and simplicity, being more amenable to embeddings in general-purpose language constructs and type systems.

In terms of expressiveness, Ferrite contributes over all prior session-based works in its support for shared session types [1], allowing it to express real-world protocols, as demonstrated by our implementation of Servo's canvas component. Shared session types reclaim the expressiveness of the untyped asynchronous π-calculus in session-typed languages [2], at the cost of deadlock-freedom. Recent extensions of classical linear logic session types contribute another approach to softening the rigidity of linear session types to support multiple client sessions and nondeterminism [35] and memory cells and nondeterministic updates [37], resp.

Our technique of a judgmental embedding opens up new possibilities for embedding type systems other than session types in Rust. Although we have demonstrated that the judgmental embedding is sufficiently powerful to encode a type system like session types, the embedding is currently *shallow*, with the implementation hardcoded to use the channels and async run-time from `tokio`. Rust comes with unique features such as affine types and lifetimes that makes it especially suited for implementing concurrency primitives, as evidenced by the wealth of channel and async run-time implementations available. One of our future goals is to explore the possibility of making Ferrite a *deep* embedding of session types in Rust, so that users can choose from multiple low-level implementations. Although deep embeddings have extensively been explored for languages like Haskell [40, 27], it remains a open question to find suitable approaches that work well in Rust.

References

1. Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP):37:1–37:29, 2017.
2. Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. A universal session type for untyped asynchronous communication. In *29th International Conference on Concurrency Theory (CONCUR)*, LIPIcs, pages 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
3. Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *28th European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019. `doi:10.1007/978-3-030-17184-1_22`.
4. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR)*, pages 222–236. Springer, 2010.
5. Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.
6. Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite project website. `https://github.com/ferrite-rs/ferrite`.
7. Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: A judgmental embedding of session types in rust. *CoRR*, abs/2009.13619, 2022. `arXiv:2009.13619`.
8. Zak Cutner and Nobuko Yoshida. Safe session-based asynchronous coordination in rust. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 80–89. Springer, 2021. `doi:10.1007/978-3-030-78142-2_5`.
9. Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. *CoRR*, abs/2112.12693, 2021. `arXiv:2112.12693`.
10. Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Principles and Practice of Declarative Programming (PPDP)*, pages 139–150, 2012.
11. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007. `doi:10.1145/1232420.1232424`.
12. Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *Proceedings of the ACM on Programming Languages*, 3(POPL):28:1–28:29, 2019. `doi:10.1145/3290341`.
13. Andrew Gerrand. The go blog: Share memory by communicating, 2010. URL: `https://blog.golang.org/share-memory-by-communicating`.
14. Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR)*, pages 509–523. Springer, 1993.
15. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP)*, pages 122–138. Springer, 1998.
16. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.
17. Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *24th European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010. `doi:10.1007/978-3-642-14107-2_16`.

18 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *22nd European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008. `doi:10.1007/978-3-540-70592-5_22`.

19 Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: a session-based library with polarities and lenses. *Science of Computer Programming*, 172:135–159, 2019. `doi:10.1016/j.scico.2018.08.005`.

20 Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in haskell. In *3rd Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES) 2010, Paphos, Cyprus, 21st March 201*, volume 69 of *EPTCS*, pages 74–91, 2010. `doi:10.4204/EPTCS.69.6`.

21 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *11th ACM SIGPLAN Workshop on Generic Programming (WGP)*, 2015. `doi:10.1145/2808098.2808100`.

22 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018. `doi:10.1145/3158154`.

23 Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, pages 96–107. ACM, 2004. `doi:10.1145/1017472.1017488`.

24 Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002. `doi:10.1007/978-3-540-40007-3_26`.

25 Wen Kokke. Rusty variation: Deadlock-free sessions with failure in rust. In *12th Interaction and Concurrency Experience, ICE 2019*, pages 48–60, 2019.

26 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *24th European Symposium on Programming (ESOP)*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584, 2015. `doi:10.1007/978-3-662-46669-8_23`.

27 Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *9th International Symposium on Haskell*, pages 133–145. ACM, 2016. `doi:10.1145/2976002.2976018`.

28 J. Garrett Morris. Variations on variants. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 71–81. ACM, 2015. `doi:10.1145/2804302.2804320`.

29 Mozilla. Servo, the Parallel Browser Engine Project. `https://servo.org/`, 2012.

30 Mozilla. Servo source code – canvas paint thread, 2021. URL: `https://github.com/servo/servo/blob/d13a9355b8e66323e666dde7e82ced7762827d93/components/canvas/canvas_paint_thread.rs`.

31 Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 568–581. ACM, 2016. `doi:10.1145/2837614.2837634`.

32 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017. `doi:10.1017/S0956796816000289`.

33 Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *Programming Journal*, 1(2):7, 2017. `doi:10.22152/programming-journal.org/2017/1/7`.

34 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *1st ACM SIGPLAN Symposium on Haskell*, pages 25–36. ACM, 2008. `doi:10.1145/1411286.1411290`.

35 Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *CoRR*, abs/2010.13926, 2020. arXiv:2010.13926.

36 John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, volume 2, pages 717–740. ACM, 1972. doi:10.1145/800194.805852.

37 Pedro Rocha and Luís Caires. Propositions-as-types and shared state. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.

38 Matthew Sackman and Susan Eisenbach. Session types in haskell: Updating message passing for the 21st century. Technical report, Imperial College, 2008. URL: http://hdl.handle.net/10044/1/5918.

39 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in Scala. In *30th European Conference on Object-Oriented Programming (ECOOP)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:28. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016.

40 Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2012. doi:10.1007/978-3-642-40447-4_2.

41 Tokio. Tokio Homepage. https://tokio.rs/, 2021.

42 Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and New University of Lisbon, 2015.

43 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *22nd European Symposium on Programming (ESOP)*, pages 350–369. Springer, 2013. doi:10.1007/978-3-642-37036-6_20.

44 Philip Wadler. Propositions as sessions. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012.

45 WebKit. MotionMark Homepage. https://browserbench.org/MotionMark/, 2021.

46 Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 119–135, 2014. doi:10.1007/978-3-319-07151-0_8.

47 Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. Ghostcell: separating permissions from data in rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473597.

48 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

A Self-Dual Distillation of Session Types

Jules Jacobs ✉
Radboud University Nijmegen, The Netherlands

Abstract

We introduce $\bar{\lambda}$ ("lambda-barrier"), a minimal extension of linear λ-calculus with concurrent communication, which adds only a *single* new **fork** construct for spawning threads. It is inspired by GV, a session-typed functional language also based on linear λ-calculus. Unlike GV, $\bar{\lambda}$ strives to be as simple as possible, and adds no new operations other than **fork**, no new type formers, and no explicit definition of session type duality. Instead, we use linear function function type $\tau_1 \multimap \tau_2$ for communication between threads, which is dual to $\tau_2 \multimap \tau_1$, *i.e.,* the function type constructor is dual to itself. Nevertheless, we can encode session types as $\bar{\lambda}$ types, GV's channel operations as $\bar{\lambda}$ terms, and show that this encoding is type-preserving. The linear type system of $\bar{\lambda}$ ensures that all programs are deadlock-free and satisfy global progress, which we prove in Coq. Because of $\bar{\lambda}$'s minimality, these proofs are simpler than mechanized proofs of deadlock freedom for GV.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages

Keywords and phrases Linear types, concurrency, lambda calculus, session types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.23

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.15
Software (Mechanized proofs): https://zenodo.org/record/6560443

Acknowledgements I thank Robbert Krebbers, Stephanie Balzer, Jorge Pérez, Dan Frumin, Bas van den Heuvel, Anton Golov, Ike Mulder, and last but not least, the anonymous reviewers for the helpful discussions and feedback.

1 Introduction

Session types [16, 15] are types for communication channels, that can be used to verify that programs follow the communication protocol specified by a channel's session type. Gay and Vasconcelos [13] embed session types in a linear λ-calculus. Whereas Gay and Vasconcelos' calculus [13] did not yet ensure deadlock freedom, Wadler's subsequent GV [31] and its derivatives [21, 23, 24, 11, 10] guarantee that all well-typed programs are deadlock free.

In order to add session types to linear λ-calculus, one adds (linear) session type formers for typing channel protocols and their corresponding operations: $\mathbf{!}\tau.s$ (send a message of type τ, continue with protocol s), $\mathbf{?}\tau.s$ (receive a message of type τ, continue with protocol s), $s_1 \oplus s_2$ (send choice between protocols s_1 and s_2), $s_1 \,\&\, s_2$ (receive choice between protocols s_1 and s_2), and End (close channel). One also adds a **fork** operation for creating a thread and a pair of dual channels. For this, we need a definition of duality, with **!** dual to **?**, $\oplus$ dual to &, and End dual to itself.

There have been efforts for simpler systems, such as an encoding of session types into ordinary π-calculus types [19, 7, 8], and *minimal session types* [2], which decompose multi-step session types into single-step session types in a π-calculus. Single-shot synchronization primitives have also been used in the implementation of a session-typed channel libraries [30, 27, 20].

We show that linear λ-calculus is also an excellent substrate on which to build a minimal concurrent calculus with communication, and introduce $\bar{\lambda}$ ("lambda-barrier"), which adds only a *single* new **fork** construct for spawning threads. It is inspired by GV, a session-typed

© Jules Jacobs;
licensed under Creative Commons License CC-BY 4.0

functional language that is also based on linear λ-calculus. Unlike GV, ƛ strives to be as simple as possible, and adds no new operations other than **fork**, no new type formers, and no explicit definition of duality. Instead, we use the linear function type $\tau_1 \multimap \tau_2$ for communication between threads, which is dual to $\tau_2 \multimap \tau_1$, *i.e.,* the function type constructor is dual to itself. Nevertheless, we can encode session types as ƛ types, GV's channel operations as ƛ terms, and show that this encoding is type-preserving. A key difference with CPS encodings of GV [22, 23], which are whole-program, is that our encoding is local, and uses ƛ's built-in concurrency.

Like GV, all well-typed ƛ programs are automatically *deadlock free*, and therefore satisfy *global progress*. We prove this property in Coq. Because of ƛ's minimality, these proofs are simpler and shorter than mechanized proofs of deadlock freedom for GV.

The rest of this article is structured as follows:

- An introduction to ƛ by example (Section 2).
- The ƛ type system and operational semantics (Section 3).
- Encoding session types in ƛ (Section 4).
- How to prove global progress and deadlock freedom for ƛ (Section 5).
- Extending ƛ with unrestricted and recursive types (Section 6).
- Mechanizing the meta-theory of ƛ in Coq (Section 7).
- Related work (Section 8).
- Concluding remarks (Section 9).

2 The ƛ language by example

The ƛ language consists of linear λ-calculus with a single extension: **fork**.[1] Let us look at an example:

$$\textbf{let } x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1)) \textbf{ in } \textbf{print}(1 + x\ 0)$$

This program forks off a new thread, which also creates *communication barriers* x and x' to communicate between the threads. The barrier x gets returned to the main thread, and x' gets passed to the child thread. These barriers are functions, and a call to a barrier will block until the other side is also trying to synchronize, and will then atomically exchange the values passed as an argument. The example runs as follows:

- When $x'\ 1$ is called, it will block until $x\ 0$ is also called, and vice versa.
- The call $x'\ 1$ will then return 0, and the call $x\ 0$ will return 1.

Thus, the program will print 0 2 or 2 0, depending on which thread prints first. In ƛ, these barriers are *linear*, so they must be used exactly once:

$$\textbf{fork}(\lambda x.\ \textbf{print}(1)) \quad \text{Error! Must use x.}$$
$$\textbf{fork}(\lambda x.\ \textbf{print}(x\ 0 + x\ 1)) \quad \text{Error! Can't use x twice.}$$

The type of **fork** is:

$$\textbf{fork} : ((\tau_1 \multimap \tau_2) \multimap \mathbf{1}) \to (\tau_2 \multimap \tau_1)$$

where $\multimap$ is the type of linear functions. Linearity allows us to encode *session types* in ƛ (Section 4), and ensures that all well-typed ƛ programs are *deadlock-free* (Section 5), which would not be the case without linearity. Nevertheless, linearity may seem like a critical

[1] For the examples we also use **print**, to be able to talk about the operational behavior of programs.

limitation: can a child thread communicate with its parent thread only *once*?! Luckily, two features of λ-calculus, namely the ability for closures to capture values from their lexical environment, and the ability to pass functions as arguments to other functions, means that the restriction is not as severe as it may seem. Let's look at an example that uses those two features:[2]

$$\textbf{let } x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1))$$
$$\textbf{let } y = \textbf{fork}(\lambda y'.\ y'\ x)$$
$$\textbf{print}(1 + y\ ()\ 0)$$

We fork off a new thread (line 1) and store its barrier in x. We then fork off another thread (line 2), and pass the barrier x into the λ-expression of the new thread. We call y (), which returns x, because the thread of y calls $y'\ x$. Finally, we pass 0 into the returned x, so this example behaves the same as the first example: it prints 0 2 or 2 0.

We can use the ability to capture barriers in the λ-expression of a fork, and the ability to send barriers over barriers, to set up long-running communication between two threads:

$$\begin{aligned}
&\textbf{let } x_1 = \textbf{fork}(\lambda x_1'.\ \textbf{let } (x_2', n_1) = x_1'\ () \\
&\qquad\qquad\qquad\quad \textbf{let } (x_3', n_2) = x_2'\ ()\ \textbf{in } x_3'\ (n_1 + n_2)) \\
&\textbf{let } x_2 = \textbf{fork}(\lambda x_2'.\ x_1\ (x_2', 1)) \\
&\textbf{let } x_3 = \textbf{fork}(\lambda x_3'.\ x_2\ (x_3', 2)) \\
&\textbf{print}(x_3\ ())
\end{aligned}$$

Let us focus on the body of the first fork. The forked thread firstly synchronizes with its barrier, via x_1' (). This call will return a pair (x_2', n_1) of a *new* barrier x_2', and a number n_1. It then synchronizes with the new barrier, via x_2' (), which returns *another* pair (x_3', n_2), giving yet another barrier x_3' and another number n_2. In the last step, it sends the number $n_1 + n_2$ back to the main thread, via $x_3'\ (n_1 + n_2)$.

Let us now focus on how the main thread arranges this sequence of communications. The main thread first forks off a *messenger thread* $\textbf{fork}(\lambda x_2'.\ x_1\ (x_2', 1))$. The purpose of this thread is to send the message $(x_2', 1)$ over x_1, where x_2' is the barrier associated with the messenger thread. The other side of that barrier, x_2, is given to the main thread. The main thread now forks off another messenger thread, this time using that new barrier, x_2. This gives the main thread yet another barrier, x_3, from which it receives the final answer, $1 + 2$, via x_3 ().

Note that, like in the asynchronous π-calculus, sending a message involves forking off a tiny thread. Thus, like the asynchronous π-calculus, $\bar{\lambda}$ should be viewed as a theoretical core calculus, and not as a practical way to implement message passing.[3] We can encapsulate this messenger thread pattern in a small library of channel operations:

$$\begin{aligned}
\textbf{send}(c, x) &\triangleq \textbf{fork}(\lambda c'.\ c\ (c', x)) \\
\textbf{receive}(c) &\triangleq c\ () \\
\textbf{close}(c) &\triangleq c\ ()
\end{aligned}$$

---

[2] We omit the **in** keyword if a newline follows **let**, like some functional languages (*e.g.,* F#).

[3] Because the messenger threads are always of a specific form, it might be possible to implement a compiler that recognizes such patterns and implements them more efficiently. After all, the messenger threads do nothing but immediately synchronize with another barrier.

Using this channel library, we can implement the preceding example in the following way:

$$\begin{array}{l}
\textbf{let } x_1 = \textbf{fork}(\lambda x'_1.\ \begin{array}[t]{l}\textbf{let } (x'_2, n_1) = \textbf{receive}(x'_1) \\ \textbf{let } (x'_3, n_2) = \textbf{receive}(x'_2) \\ \textbf{let } x'_4 = \textbf{send}(x'_3, n_1 + n_2) \\ \textbf{close}(x'_4))\end{array} \\
\textbf{let } x_2 = \textbf{send}(x_1, 1) \\
\textbf{let } x_3 = \textbf{send}(x_2, 2) \\
\textbf{let } (x_4, n) = \textbf{receive}(x_3) \\
\textbf{print}(n) \\
\textbf{close}(x_4)
\end{array}$$

Session-typed channels usually also have *choice*, which allows choosing between two continuation protocols. This can be encoded in ƛ using sums $\textbf{in}_{\mathbf{L}}(x)$ and $\textbf{in}_{\mathbf{R}}(x)$:

$$\begin{aligned}
\textbf{tell}_{\mathbf{L}}(c) &\triangleq \textbf{fork}(\lambda c'.\ c\ \textbf{in}_{\mathbf{L}}(c')) \\
\textbf{tell}_{\mathbf{R}}(c) &\triangleq \textbf{fork}(\lambda c'.\ c\ \textbf{in}_{\mathbf{R}}(c')) \\
\textbf{ask}(c) &\triangleq c\ ()
\end{aligned}$$

With these operations, we can implement the calculator example of Lindley and Morris [24]. This example allows the client to choose whether they want to add two numbers or negate a number. If the client chooses to add two numbers, they then send two numbers as separate messages, and retrieve the answer using receive. If the client chooses to negate a number, they then send only a single number, and retrieve the answer using receive. This example illustrates the choice between two different protocols for the remaining interaction:

$$\begin{array}{l}
\textbf{let } \mathsf{calc}\ c = \\
\quad \textbf{match ask}(c) \textbf{ with} \\
\quad \mid \textbf{in}_{\mathbf{L}}(c) \Rightarrow \begin{array}[t]{l}\textbf{let } (c, n) = \textbf{receive}(c) \\ \textbf{let } (c, m) = \textbf{receive}(c) \\ \textbf{close}(\textbf{send}(c, n + m))\end{array} \\
\quad \mid \textbf{in}_{\mathbf{R}}(c) \Rightarrow \begin{array}[t]{l}\textbf{let } (c, n) = \textbf{receive}(c) \\ \textbf{close}(\textbf{send}(c, -n))\end{array} \\
\quad \textbf{end}
\end{array}$$

Extending ƛ with recursion (Section 6) allows us to implement unbounded protocols, as illustrated by the following example:

$$\begin{array}{l}
\textbf{let rec } \mathsf{countdown}\ c = \\
\quad \textbf{match ask}(c) \textbf{ with} \\
\quad \mid \textbf{in}_{\mathbf{L}}(c) \Rightarrow \textbf{close}(c) \\
\quad \mid \textbf{in}_{\mathbf{R}}(c) \Rightarrow \begin{array}[t]{l}\textbf{let } (c, n) = \textbf{receive}(c) \\ \textbf{print}(n) \\ \textbf{if } n = 0 \\ \ \textbf{then close}(\textbf{tell}_{\mathbf{L}}(c)) \\ \ \textbf{else } \mathsf{countdown}\ (\textbf{send}(\textbf{tell}_{\mathbf{R}}(c), n - 1))\end{array} \\
\quad \textbf{end}
\end{array}$$

Given a channel c, the $\mathsf{countdown}\ c$ program first uses $\textbf{ask}(c)$ to ask c if it wants to terminate, and closes the channel if so. Otherwise it receives a number n from c and prints it. Then it checks if the number $n = 0$, and if so tells the other side to close (using $\textbf{tell}_{\mathbf{L}}$),

and then closes our side. Otherwise it tells the other side that it wants to continue (using $\mathbf{tell_R}$) and sends $n-1$ to the other side. We can therefore let countdown interact with a copy of itself, provided we start off one of the copies with an initial message:

$$\mathsf{countdown}\ \mathbf{send}(\mathbf{tell_R}(\mathbf{fork}(\mathsf{countdown})), 10)$$

This program will print the numbers 10 9 8 $\cdots$ 1 0, in that order. The odd numbers are printed by the main thread, and the even numbers are printed by the child thread.

As we shall see later, this is all type-safe. If we had not started off one of the countdowns with an initial message, and had instead simply done countdown **fork**(countdown), then we would have had a static type error.

This way, the $\lambda\!\!\!\!\lambda$ type system ensures that channel protocols are correctly followed, even though the $\lambda\!\!\!\!\lambda$ type system has no session types and no notion of duality and instead simply uses function types $\tau_1 \multimap \tau_2$ for barriers. We do not need an explicit notion of duality because the function type constructor is *self-dual*, in the sense that if $x : \tau_1 \multimap \tau_2$ is a barrier, then the dual barrier with which x will synchronize has type $x' : \tau_2 \multimap \tau_1$. We will see more about encoding session types in $\lambda\!\!\!\!\lambda$ in Section 4.

3 The $\lambda\!\!\!\!\lambda$ type system and operational semantics

Like GV [31] and its derivatives [21, 23, 24, 11, 10], the basis of $\lambda\!\!\!\!\lambda$ is a linear simply typed λ-calculus. We have sums, products, and the linear function type $\tau_1 \multimap \tau_2$. Variables of linear type must be used exactly once: they cannot be duplicated (contracted) or discarded (weakened), so that one must use one of the elimination rules of the type.

$$\tau \in \mathit{Type} ::= \mathbf{0} \mid \mathbf{1} \mid \tau \times \tau \mid \tau + \tau \mid \tau \multimap \tau$$

Our basis linear λ-calculus has the following grammar of expressions, which consists of introduction and elimination forms for each type:

$$\begin{aligned} e \in \mathit{Expr} ::= {} & x \mid () \mid (e,e) \mid \mathbf{in_L}(e) \mid \mathbf{in_R}(e) \mid \lambda x.\ e \mid e\ e \mid \mathbf{let}\ (x_1,x_2) = e\ \mathbf{in}\ e \mid \\ & \mathbf{match}\ e\ \mathbf{with}\ \ \mathbf{end} \mid \mathbf{match}\ e\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} \end{aligned}$$

The typing rules are standard and can be found in Figure 1.

We now have a substrate to which we will add concurrency constructs. GV [31, 21, 23, 24, 11, 10] introduces concurrency by means of a construct to spawn a new thread, with which we can communicate using a channel. Communication is governed by session types, such that the two endpoints of a channel are typed with dual session types. Instead, $\lambda\!\!\!\!\lambda$ extends the substrate with concurrency in a minimal way, adding one new construct to create new threads:

$$e \in \mathit{Expr} ::= \cdots \mid \mathbf{fork}(e)$$

This is the typing rule for **fork**:

$$\frac{\Gamma \vdash e : (\tau_2 \multimap \tau_1) \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}(e) : \tau_1 \multimap \tau_2}$$

Or, as a type signature:

$$\mathbf{fork} : ((\tau_1 \multimap \tau_2) \multimap \mathbf{1}) \to (\tau_2 \multimap \tau_1)$$

$$\frac{\cdot}{x{:}\tau \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \textbf{let}\ (x_1, x_2) = e_1\ \textbf{in}\ e_2 : \tau_3}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{in}_\mathbf{L}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{in}_\mathbf{R}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma_1 \vdash e : \tau_1 + \tau_2 \qquad \Gamma_2, x_1{:}\tau_1 \vdash e_1 : \tau' \qquad \Gamma_2, x_2{:}\tau_2 \vdash e_2 : \tau'}{\Gamma_1, \Gamma_2 \vdash \textbf{match}\ e\ \textbf{with}\ \textbf{in}_\mathbf{L}(x_1) \Rightarrow e_1;\ \textbf{in}_\mathbf{R}(x_2) \Rightarrow e_2\ \textbf{end} : \tau'}$$

Figure 1 Linear λ-calculus with sums and products (rules for **0** and **1** omitted).

The type of **fork** uses the linear function type. We do not need an explicit notion of duality, like session types do, because the function type constructor is *self-dual*, in the sense that if $x : \tau_1 \multimap \tau_2$ is a barrier, then the dual barrier x' with which x will synchronize has type $x' : \tau_2 \multimap \tau_1$.

3.1 Operational semantics

We use a small-step operational semantics with evaluation contexts. In order to represent barriers, we add barrier literals $\langle k \rangle, k \in \mathbb{N}$ to the expressions. A barrier literal cannot appear in the source program, as the static type system has no typing rule for it. Barrier literals only appear in expressions at runtime when the operational semantics executes a **fork**-step. This gives us the following set of values for the language:

$$v \in \mathit{Val} ::= () \mid (v, v) \mid \textbf{in}_\mathbf{L}(v) \mid \textbf{in}_\mathbf{R}(v) \mid \lambda x.\ e \mid \langle k \rangle$$

We have four pure head-reduction rules $e \rightsquigarrow_{\text{pure}} e'$, one for λ, one for pairs, and two for sums, as stated in Figure 2. We use evaluation contexts to avoid introducing many congruence rules:[4]

$$\begin{aligned} K ::= {} & \square \mid (K, e) \mid (v, K) \mid \textbf{in}_\mathbf{L}(K) \mid \textbf{in}_\mathbf{R}(K) \mid K\ e \mid v\ K \mid \textbf{fork}(K) \mid \textbf{let}\ (x_1, x_2) = K\ \textbf{in}\ e \\ & \mid \textbf{match}\ K\ \textbf{with}\ \ \textbf{end} \mid \textbf{match}\ K\ \textbf{with}\ \textbf{in}_\mathbf{L}(x_1) \Rightarrow e_1;\ \textbf{in}_\mathbf{R}(x_2) \Rightarrow e_2\ \textbf{end} \end{aligned}$$

We represent a configuration with multiple threads and barriers as a finite map:

$$\rho \in \mathit{Cfg} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \mathsf{Thread}(\mathit{Expr}) + \mathsf{Barrier}$$

We define a **configuration step relation** $\rho \overset{i}{\rightsquigarrow} \rho'$. The label $i \in \mathbb{N}$ is used to keep track of which thread or barrier in the configuration takes the step. This has no effect on the operational semantics, but we will later use it to formulate deadlock freedom. The rules for the configuration step relation are given in Figure 2. We have the following five rules, in the order of the figure:

[4] This set of evaluation contexts results in left-to-right evaluation order, but the mechanization has been set up so that the proof scripts work for right-to-left and nondeterministic order as well.

$$(\lambda x.\ e)\ v \rightsquigarrow_{\text{pure}} e[v/x]$$
$$\textbf{let}\ (x_1, x_2) = (v_1, v_2)\ \textbf{in}\ e \rightsquigarrow_{\text{pure}} e[v_1/x_1][v_2/x_2]$$
$$\textbf{match}\ \textbf{in}_\textbf{L}(v)\ \textbf{with}\ \textbf{in}_\textbf{L}(x_1) \Rightarrow e_1 \mid \textbf{in}_\textbf{R}(x_2) \Rightarrow e_2\ \textbf{end} \rightsquigarrow_{\text{pure}} e_1[v/x_1]$$
$$\textbf{match}\ \textbf{in}_\textbf{R}(v)\ \textbf{with}\ \textbf{in}_\textbf{L}(x_1) \Rightarrow e_1 \mid \textbf{in}_\textbf{R}(x_2) \Rightarrow e_2\ \textbf{end} \rightsquigarrow_{\text{pure}} e_2[v/x_2]$$

$$\{n \mapsto \mathsf{Thread}(K[\,e_1\,])\} \overset{n}{\rightsquigarrow} \{n \mapsto \mathsf{Thread}(K[\,e_2\,])\} \quad \text{if } e_1 \rightsquigarrow_{\text{pure}} e_2 \qquad \text{(pure)}$$

$$\{n \mapsto \mathsf{Thread}(K[\,\textbf{fork}(v)\,])\} \overset{n}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \langle k\rangle\ ]) \\ k \mapsto \mathsf{Barrier} \\ m \mapsto \mathsf{Thread}(v\ \ \langle k\rangle) \end{Bmatrix} \qquad \text{(fork)}$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K_1[\ \langle k\rangle\ \ v_1\,]) \\ k \mapsto \mathsf{Barrier} \\ m \mapsto \mathsf{Thread}(K_2[\ \langle k\rangle\ \ v_2\,]) \end{Bmatrix} \overset{k}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K_1[\,v_2\,]) \\ m \mapsto \mathsf{Thread}(K_2[\,v_1\,]) \end{Bmatrix} \qquad \text{(sync)}$$

$$\{n \mapsto \mathsf{Thread}(())\} \overset{n}{\rightsquigarrow} \{\} \qquad \text{(exit)}$$

$$\rho_1 \uplus \rho' \overset{i}{\rightsquigarrow} \rho_2 \uplus \rho' \quad \text{if } \rho_1 \overset{i}{\rightsquigarrow} \rho_2 \quad (\uplus \text{ is disjoint union}) \qquad \text{(frame)}$$

Figure 2 The operational semantics of ƛ.

(pure) A rule for pure reductions for a single thread.
(fork) A rule for forking a new thread, which adds the new thread and a barrier k to the configuration. The two threads get access to the barrier via the barrier literal $\langle k\rangle$.
(sync) A rule to synchronize on a barrier. The two threads that are synchronizing exchange the values v_1 and v_2. This step removes the barrier.
(exit) A rule for removing finished threads from the configuration.
(frame) A rule for extending the preceding rules to larger configurations, by allowing the rest of the configuration to pass through unchanged.

This is a possible execution of the second example from Section 2:

$$\left\{ 0 \mapsto \mathsf{Thread} \begin{pmatrix} \textbf{let}\ x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1)) \\ \textbf{let}\ y = \textbf{fork}(\lambda y'.\ y'\ x) \\ \textbf{print}(1 + y\ ()\ 0) \end{pmatrix} \right\} \overset{0}{\rightsquigarrow}$$

$$\begin{Bmatrix} 0 \mapsto \mathsf{Thread} \begin{pmatrix} \textbf{let}\ y = \textbf{fork}(\lambda y'.\ y'\ \ \langle 1\rangle) \\ \textbf{print}(1 + y\ ()\ 0) \end{pmatrix} \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \end{Bmatrix} \overset{0}{\rightsquigarrow} \begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + \langle 3\rangle\ \ ()\ 0)) \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \\ 3 \mapsto \mathsf{Barrier} \\ 4 \mapsto \mathsf{Thread}\,(\langle 3\rangle\ \ \langle 1\rangle) \end{Bmatrix} \overset{3}{\rightsquigarrow}$$

$$\begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + \langle 1\rangle\ \ 0)) \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \\ 4 \mapsto \mathsf{Thread}\,(()) \end{Bmatrix} \overset{4}{\rightsquigarrow} \begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + \langle 1\rangle\ \ 0)) \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \end{Bmatrix} \overset{1}{\rightsquigarrow}$$

$$\begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + 1)) \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(0)) \end{Bmatrix} \overset{2*}{\rightsquigarrow} \{0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + 1))\} \overset{1*}{\rightsquigarrow} \{\}$$

For simplicity, we treat **print** on natural numbers as a no-op that returns (), instead of adding an output log to the semantics, because whether or not **print** logs its output somewhere does not affect the further execution of the program.

While untyped ƛ programs can easily get stuck, for instance if one side throws away its barrier, or sets up cyclic waiting dependencies, well-typed ƛ programs never get stuck. More formally, we prove *global progress* (in Section 5), which means that if we start with an initial program $e : \mathbf{1}$, then any non-empty configuration we can reach from e can step further. But first, we will see how to encode session types in ƛ.

4 Encoding session types in ƛ

Despite being very simple, ƛ's type system can encode session types. There are five basic session type constructors:

$$s \in \mathit{Session} \triangleq \mathsf{End} \mid \mathbf{!}\tau.s \mid \mathbf{?}\tau.s \mid s \oplus s \mid s \,\&\, s$$

The type $\mathbf{!}\tau.s$ means to send a value of type τ and then continue with s. Dually, $\mathbf{?}\tau.s$ means to receive a value of type τ and then continue with s. The type $s_1 \oplus s_2$ indicates that we have a choice of continuing either with protocol s_1 or with s_2. Dually, the type $s_1 \,\&\, s_2$ means that we receive a choice from the other side: either we have to continue with protocol s_1 or with protocol s_2, depending on what the other side chose. Lastly, the protocol End means that we are done with the channel and we must deallocate it. Session types make the notion of duality explicit using the function $\mathsf{dual} : \mathit{Session} \to \mathit{Session}$:

$$\begin{aligned}
\mathsf{dual}(\mathsf{End}) &\triangleq \mathsf{End}\\
\mathsf{dual}(\mathbf{!}\tau.s) &\triangleq \mathbf{?}\tau.\mathsf{dual}(s)\\
\mathsf{dual}(\mathbf{?}\tau.s) &\triangleq \mathbf{!}\tau.\mathsf{dual}(s)\\
\mathsf{dual}(s_1 \oplus s_2) &\triangleq \mathsf{dual}(s_1) \,\&\, \mathsf{dual}(s_2)\\
\mathsf{dual}(s_1 \,\&\, s_2) &\triangleq \mathsf{dual}(s_1) \oplus \mathsf{dual}(s_2)
\end{aligned}$$

The idea is that if our channel has type s, then the channel of the party we are communicating with has type $\mathsf{dual}(s)$. This is the list of channel operations and their types:

fork$_{GV}$: $(s \multimap 1) \to \mathsf{dual}(s)$
Fork off a new thread running the closure. Passes a channel of type s to the child thread, and returns a channel of type $\mathsf{dual}(s)$ to the main thread.

close : $\mathsf{End} \to 1$
Close and deallocate the channel. Returns unit ().

send : $\mathbf{!}\tau.s \times \tau \to s$
Send a message of type τ to the channel. Returns a new channel of type s for performing the rest of the protocol.

receive : $\mathbf{?}\tau.s \to s \times \tau$
Receive a message from the channel. Returns a pair $s \times \tau$ of the channel for performing the rest of the protocol (type s) and the message received (type τ).

tell$_{\mathrm{L}}$: $s_1 \oplus s_2 \to s_1$
In a branching protocol, choose the left branch. Returns a channel of the chosen type.

tell$_{\mathrm{R}}$: $s_1 \oplus s_2 \to s_2$
In a branching protocol, choose the right branch. Returns a channel of the chosen type.

ask : $s_1 \mathbin{\&} s_2 \to s_1 + s_2$
Receives the choice made by the other side. Returns a sum type, which is $\mathbf{in_L}(c)$ with $c : s_1$ if the left branch was chosen by the other side, and $\mathbf{in_R}(c)$ with $c : s_2$ if the right branch was chosen.[5]

We will encode channels as $\lambda\!\!\!\!-$'s barriers, and we therefore encode a session type s as a linear function type $\tau_1 \multimap \tau_2$ where τ_1, τ_2 are determined from s. Intuitively, the sending side not only transfers the values specified by the protocol, but also a continuation channel for the remainder of the protocol. The continuation channel is connected to a tiny messenger thread, which is responsible for synchronizing with the old barrier, as we did in Section 2. We define an encoding function $[\![\cdot]\!] : \mathit{Session} \to \mathit{Type}$ that converts a session type to a $\lambda\!\!\!\!-$ type. The encoding of session types into $\lambda\!\!\!\!-$ types is as follows:

$$
\begin{aligned}
[\![\mathsf{End}]\!] &\triangleq \mathbf{1} \multimap \mathbf{1} \\
[\![!\tau.s]\!] &\triangleq [\![\mathsf{dual}(s)]\!] \times \tau \multimap \mathbf{1} \\
[\![?\tau.s]\!] &\triangleq \mathbf{1} \multimap [\![s]\!] \times \tau \\
[\![s_1 \oplus s_2]\!] &\triangleq [\![\mathsf{dual}(s_1)]\!] + [\![\mathsf{dual}(s_2)]\!] \multimap \mathbf{1} \\
[\![s_1 \mathbin{\&} s_2]\!] &\triangleq \mathbf{1} \multimap [\![s_1]\!] + [\![s_2]\!]
\end{aligned}
$$

Using this encoding, we can implement channel operations with type signatures matching their native session-typed version, provided we use the encoding:

$$
\begin{aligned}
\mathbf{fork}_{GV} &: ([\![s]\!] \multimap \mathbf{1}) \to [\![\mathsf{dual}(s)]\!] & \mathbf{fork}_{GV}(x) &\triangleq \mathbf{fork}(x) \\
\mathbf{close} &: [\![\mathsf{End}]\!] \to \mathbf{1} & \mathbf{close}(c) &\triangleq c\ () \\
\mathbf{send} &: [\![!\tau.s]\!] \times \tau \to [\![s]\!] & \mathbf{send}(c, x) &\triangleq \mathbf{fork}(\lambda c'.\ c\ (c', x)) \\
\mathbf{receive} &: [\![?\tau.s]\!] \to [\![s]\!] \times \tau & \mathbf{receive}(c) &\triangleq c\ () \\
\mathbf{tell_L} &: [\![s_1 \oplus s_2]\!] \to [\![s_1]\!] & \mathbf{tell_L}(c) &\triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_L}(c')) \\
\mathbf{tell_R} &: [\![s_1 \oplus s_2]\!] \to [\![s_2]\!] & \mathbf{tell_R}(c) &\triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_R}(c')) \\
\mathbf{ask} &: [\![s_1 \mathbin{\&} s_2]\!] \to [\![s_1]\!] + [\![s_2]\!] & \mathbf{ask}(c) &\triangleq c\ ()
\end{aligned}
$$

The fork operation for channels simply delegates to the fork operation of $\lambda\!\!\!\!-$, because a channel is represented as a barrier.

Formal statement of well-typedness of the encodings

You may note that while there is an encoding function $[\![\cdot]\!]$ of session types into $\lambda\!\!\!\!-$ types, there is no explicit encoding function of GV terms to $\lambda\!\!\!\!-$ terms. This is intentional: because the translation is *local*, the definitions above can be viewed as *syntactic abbreviations* or *macros*. For instance, we can define the abbreviation

$$\mathbf{tell_L} \triangleq \lambda c.\ \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_L}(c'))$$

[5] In the original GV, branching was combined with receiving a choice. We decouple them, and let receiving a choice return a sum type, which can subsequently be pattern matched on using **match**.

of the $\mathbf{tell_L}$ channel operation as a closed syntactic ⅄ term. We can then *prove* that for all session types s_1 and s_2, the typing judgement

$$\emptyset \vdash \mathbf{tell_L} : [\![s_1 \oplus s_2]\!] \to [\![s_1]\!]$$

for the $\mathbf{tell_L}$ term given above is derivable from ⅄'s typing rules. The most interesting case is **fork**; in order to prove

$$\emptyset \vdash \mathbf{fork} : ([\![s]\!] \multimap \mathbf{1}) \to [\![\mathsf{dual}(s)]\!]$$

for all session types s, we rely on the following lemma about dual and the encoding $[\![\cdot]\!]$:

$$([\![s]\!] = \tau_1 \multimap \tau_2) \iff ([\![\mathsf{dual}(s)]\!] = \tau_2 \multimap \tau_1)$$

That is, if the session types are dual in the session types sense, then their encodings are dual in the ⅄ sense.

The advantage of this approach is its simplicity and that we can freely intermix channels with direct usage of barriers in the same program. However, for our simulation result (Section 4.1), we do need an explicit definition of GV syntax and a translation of GV terms to ⅄ terms.

A note on the mechanization and n-ary choice

We can combine n-ary choice with sending/receiving a message in a single communication step using n-ary sum types:

$$\begin{aligned} \mathbf{sendchoice_i} &: \mathbf{!}\{i\colon \tau_i.s_i\}_{i\in I} \times \tau_i \to s_i \\ \mathbf{receivechoice} &: \mathbf{?}\{i\colon \tau_i.s_i\}_{i\in I} \to \Sigma_{i\in I}\, s_i \times \tau_i \end{aligned}$$

This can also be encoded in ⅄, and is what is provided by the mechanization (7):

$$\begin{aligned} \mathbf{sendchoice_i}(c, x) &\triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_i}(c', x)) \\ \mathbf{receivechoice}(c) &\triangleq c\ () \end{aligned}$$

Encoding ⅄ in GV

We can also do the encoding the other way around, and implement ⅄'s fork in terms of GV's channel constructs:

$$\begin{aligned}
&\mathbf{fork}_{⅄}(f) \triangleq \\
&\quad \mathbf{let}\ c_1 = \mathbf{fork}_{GV}(\lambda c_1'.\ f\ (\lambda v'.\ \mathbf{let}\ c_2' = \mathbf{send}(c_1', v') \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ (c_3', v) = \mathbf{receive}(c_2') \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{close}(c_3');\ v)) \\
&\quad \lambda v.\ \mathbf{let}\ (c_2, v') = \mathbf{receive}(c_1) \\
&\qquad \mathbf{let}\ c_3 = \mathbf{send}(c_2, v) \\
&\qquad \mathbf{close}(c_3);\ v'
\end{aligned}$$

Given how short the encodings of GV's channel operations in ⅄ are, it is perhaps surprising that the other way around requires comparatively more code.

4.1 Simulation of GV's semantics with ƛ's semantics

To show that the encoding makes sense, we prove that we can simulate an asynchronous version of GV using ƛ's semantics. We use an asynchronous semantics ($\rightsquigarrow_{GV}$) for GV, so the GV configuration contains buffers. For details about the GV semantics used in the proof, we refer the reader to the mechanization (Section 7). The key idea behind the simulation is that each message in a buffer on the GV side corresponds to a messenger thread on the ƛ side. Whenever a message is put in a buffer on the GV side, a messenger thread is created on the ƛ side, and the messenger thread will be waiting to synchronize with a barrier. Whenever a message is received from a channel's buffer on the GV side, the receiver and the messenger thread execute their sync operation on the ƛ side, which sends the message to the receiver and allows the messenger thread to terminate.

Formally, we start with an encoding $[\![e]\!]$ that translates GV terms to the corresponding ƛ terms by replacing all occurrences of GV channel operations with their ƛ definitions given above. We then extend this translation to configurations $[\![\rho]\!]$, The translation on configurations replaces each buffer in the GV heap with a sequence of ƛ messenger threads, with one messenger thread per message in the buffer. With these notions at hand, we can show that the ƛ encodings simulate the GV semantics.

▶ **Lemma 1** (Simulation). *If $\rho \rightsquigarrow_{GV} \rho'$ then $[\![\rho]\!] \rightsquigarrow^* [\![\rho']\!]$*

This lemma has been mechanized in Coq (Section 7). We need the transitive closure ($\rightsquigarrow^*$) on the ƛ side, because a single step in the GV semantics can correspond to multiple steps in the ƛ semantics, since the ƛ semantics does extra administrative β-reductions. For instance, when the GV program does a $\mathbf{send}(c, v)$ operation, it places the message v in the buffer in one step. The translated ƛ program on the other hand spawns a messenger thread with $[\![\mathbf{send}(c, v)]\!] = \mathbf{fork}(\lambda c'.\ c\ (c', v))$, which initializes the new thread with the term $(\lambda c'.\ c\ (c', v))\ \langle k\rangle$ where $\langle k\rangle$ is the newly created barrier. The messenger thread then performs the β-reduction, resulting in an extra step on the ƛ side.

To get a full operational correspondence [14], we need a second "reflection" lemma stating that if the image of the translation $[\![\rho]\!]$ takes a step, then this step can be matched with a corresponding step in the GV semantics. Note that this only holds for well-typed terms: if we have the ill-typed term $\mathbf{receive}(\lambda x.\ x)$ in the GV source, this is translated to well-typed $(\lambda x.\ x)()$ in ƛ. Thus, whereas $\mathbf{receive}(\lambda x.\ x)$ gets stuck in the GV semantics, its translation does not get stuck in the ƛ semantics. We do expect a full operational correspondence to hold for well-typed terms, but while we have mechanized the proof of the simulation direction (Lemma 1), we have not mechanized a full operational correspondence. With a full operational correspondence it would be possible to lift ƛ's deadlock freedom result to GV, and it would be interesting to see whether using ƛ as a "proof IR" in this manner is a viable strategy for proving deadlock freedom of GV.

4.2 Summary

To add GV's session types to linear λ-calculus, we need to add the 5 new session type formers, the notion of duality, and the 7 session type operations. In contrast, ƛ only adds one new operation, **fork**, and no new type formers and no notion of duality. Nevertheless, we have seen that we can encode session types in ƛ.

The encoding creates a new thread to store each message. Thus, ƛ should not be viewed as a practical way to implement session types, but as a theoretical calculus, like other calculi that create one thread per message, such as the asynchronous π-calculus.

5 Deadlock freedom, leak freedom, and global progress

Linear typing in $\bar{\lambda}$ guarantees strong properties for well-typed programs:

Type safety: threads never get stuck, except by synchronizing with a barrier.
Global progress: a non-empty configuration can always take a step.
Strong deadlock freedom: no subset of the threads gets stuck by waiting for each other.
Memory leak freedom: all barriers in the configuration remain referenced by a thread.

These properties are all inequivalent in strength: none of the 4 properties is strictly stronger than another. In Section 5.3 we consider a property that is strictly stronger than these 4, but we will first focus on *global progress* (Section 5.1), as ideas behind the proof of global progress (Section 5.2), are also sufficient to prove the stronger property (Section 5.3).

5.1 Global progress

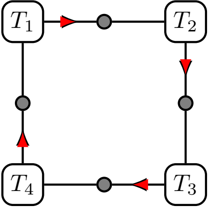
Let us consider the formal statement of global progress:

▶ **Theorem 2** (Global progress).
If $\emptyset \vdash e : \mathbf{1}$, *and* $\{0 \mapsto \mathsf{Thread}(e)\} \rightsquigarrow^* \rho$, *then either* $\rho = \{\}$ *or* $\exists \rho'. \rho \rightsquigarrow \rho'$.

Intuitively, global progress states that if we start with a well-typed program, then any configuration we reach is either empty (*i.e.,* all threads have terminated and all barriers have been deallocated), or the configuration can perform a step. This property relies on linear typing, as $\bar{\lambda}$ programs that violate linearity can deadlock and create a non-empty configuration that cannot step. A simple example is if one side does not use its barrier:

$\mathbf{let}\ x = \mathbf{fork}(\lambda x'.\ ())\ \mathbf{in}\ x\ 0$ Deadlock!

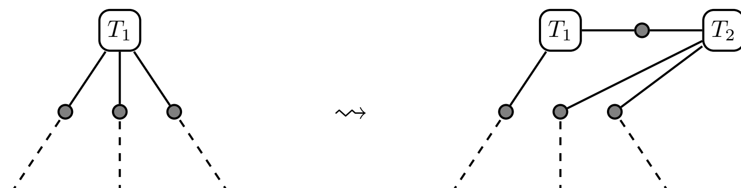
This deadlock is prevented by the linear type system, which ensures that each barrier is used *exactly once*. More complicated deadlocks are also possible in untyped programs, in which there is a circle of threads T_1 T_2 T_3 T_4 connected by barriers ● ● ● ● that are trying to synchronize (—▶—) in a cycle:



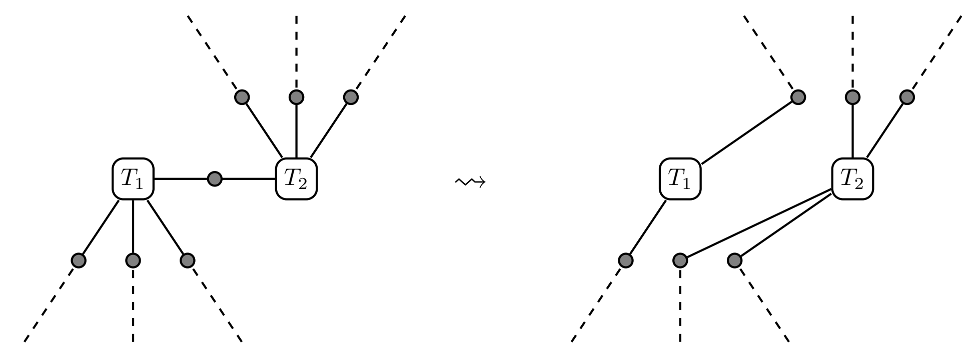
No thread can make progress because the threads are all synchronizing on different barriers. This shows that a simplistic scheme to prove deadlock freedom cannot work: we must somehow rule out such cycles. Fortunately, the linear type system ensures that the graph of connections between threads has the shape of a *forest* (*i.e.,* collection of trees, *i.e.,* an acyclic graph), and thus such circular deadlocks cannot happen. To see why the graph remains acyclic, consider what happens when we **fork**:

$\mathbf{let}\ x_1 = \mathbf{fork}(...)$
$\mathbf{let}\ x_2 = \mathbf{fork}(...)$
$\mathbf{let}\ x_3 = \mathbf{fork}(...)$
$\mathbf{let}\ y = \mathbf{fork}(\lambda y'.\ \cdots x_2 \cdots x_3 \cdots)$
$\cdots x_1 \cdots$

At the fourth **fork**, the barriers x_2 and x_3 are transferred to the new thread via lexical scoping, while the main thread keeps x_1 for itself. This is what happens to the graph:



On the left hand side, we have the thread T_1 that is about to perform the fourth **fork**. It currently owns 3 barriers x_1, x_2, x_3, which are connected to the rest of the graph. After the fork, we have the new thread T_2, which is connected to T_1 by means of a new barrier. Crucially, the barriers x_2 and x_3 of the barriers that T_1 used to own were transferred to T_2 by means of lexical scoping. Nevertheless, as one can see in the figure above, if the graph of the configuration before the fork was acyclic, then the graph of the configuration after the fork is also acyclic. The same applies to a synchronization step, when values containing potentially multiple barriers are exchanged:



On the left, T_1 and T_2 each own 3 barriers, and they are also connected by a barrier. On the right, after the synchronization has taken place, the barrier between them has disappeared. Two of the barriers of T_1 were transferred to T_2, and one of the barriers of T_2 was transferred to T_1. Once again, if the graph on the left was acyclic, then the graph on the right will still be acyclic.[6]

The other operations of ƛ do not change the connections in the graph. Therefore, a program starts with a single thread, and then grows and alters its graph in a dance of fork and sync steps, but the graph remains acyclic at all times.

In the next section we will see in a bit more detail how the acyclicity of the graph is used in the proof of global progress.

Related work. Graph theoretic deadlock-freedom arguments are common in the session types literature and have previously been made by Carbone [5], Lindley and Morris [21], and Fowler, Kokke, Dardha, Lindley, and Morris [10].

[6] When one looks at the picture of the synchronizing threads, it seems that T_1 becomes totally disconnected from T_2 after the synchronization. This means that the threads can only communicate *once*. Yet the session-typed channel encoding seems to make it possible to communicate several times. Exercise for the reader: what is the solution to this paradox?

$$\frac{\cdot}{k:\tau;\emptyset \vdash \langle k \rangle : \tau} \qquad \frac{\cdot}{\emptyset; x:\tau \vdash x : \tau} \qquad \frac{\Sigma;\Gamma, x:\tau_1 \vdash e : \tau_2}{\Sigma;\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2}$$

$$\frac{\Sigma_1;\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Sigma_2;\Gamma_2 \vdash e_2 : \tau_1}{\Sigma_1,\Sigma_2;\Gamma_1,\Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

Figure 3 Run-time type system (selected rules).

5.2 Structure of the global progress proof

This section gives more detail about how the acyclicity of the connection graph is used to prove global progress. For the full details, the interested reader is referred to the mechanization (Section 7).

Global progress states that if the configuration is non-empty, then it can take a step. Formally, there are several types of stuck configurations to rule out. Perhaps the configuration has a single thread and no barriers, but the thread is stuck on a type error (violating type safety). Or perhaps the configuration consists of just one barrier and no threads (violating memory leak freedom). Or perhaps there is a single thread and a single barrier, and the thread is trying to synchronize but is stuck due to the absence of a partner to synchronize with (violating deadlock freedom).

The aforementioned stuck configurations and more complicated variations are ruled out by keeping track of sufficient type information and local invariants for each object in the configuration (*e.g.,* that a barrier is always connected to exactly two threads, and that the types of the references to the barrier are dual $\tau_1 \multimap \tau_2$ and $\tau_2 \multimap \tau_1$). The interesting case is when the types are all locally correct, but the configuration is still deadlocked due to cyclic waiting. This case is ruled out by proving that well-typed programs maintain the invariant that the connection graph is acyclic, which implies that cyclic waiting cannot happen.

Formally, we define a *well-formedness* invariant of configurations, that maintains well-typedness of the threads as well as the acyclicity of the connection graph. For the well-typedness, we use the run-time type system in Figure 3.[7] The rules of the run-time type system correspond to the rules of the static type system, plus one additional rule for typing barrier literals $\langle k \rangle$. The typing judgment uses an additional Σ-context for typing those barrier literals.

We say that a configuration ρ is *well-formed* if we have an *acyclic* graph G (*i.e.,* an undirected forest) where the vertices correspond to the entries of the configuration, and the edges (between threads and barriers) are labeled with types, such that for each vertex i in G:

- If $\rho(i) = \mathsf{Thread}(e)$ then $\Sigma;\emptyset \vdash e : \mathbf{1}$, where Σ is given by the types on the edges of the barriers connected to vertex i.
- If $\rho(i) = \mathsf{Barrier}$ then vertex i has edges to two different threads, labeled with dual types $\tau_1 \multimap \tau_2$ and $\tau_2 \multimap \tau_1$.

These conditions ensure that the graph structure matches the structure of the configuration: if we have a barrier literal $\langle k \rangle$ somewhere in the expression e of thread n, then the first condition ensures that we have an edge between n and k, labeled with a type $\tau_1 \multimap \tau_2$

[7] Our run-time type system in Coq makes use of separation logic, following [18]. This is equivalent to the type system in the figure, but easier to work with in a proof assistant.

to make expression e well-typed. The second condition then ensures that there is a second thread with barrier literal $\langle k \rangle$ in its expression, with type $\tau_2 \multimap \tau_1$. Note that the two occurrences of the very same barrier literal $\langle k \rangle$ have two different types in the two different threads.

Now that the configuration invariant has been defined, proof of global progress (Theorem 2) can be structured as follows. Using well-typedness, we are able to show that the only way a configuration can get stuck is if all threads are trying to synchronize with a barrier. The invariant maintains that the graph structure is always acyclic. By a mathematical argument about graphs and the pigeonhole principle, we are then able to show that two of the threads must be synchronizing on the same barrier. Hence, at least one synchronization step can proceed, and we have global progress.

5.3 Strengthened deadlock and memory leak freedom

Global progress (Theorem 2) rules out whole-program deadlocks. It also ensures that all barriers have been used when the program finishes. However, this theorem does not guarantee anything as long as there is still a single thread that can step. Thus it does not guarantee local deadlock freedom, nor memory leak freedom while the program is still running, and it does not even guarantee type safety: a situation in which a thread is stuck on a type error is formally not ruled out by this theorem as long as there is another thread that can still step.

Our goal is to find a formulation that is strictly stronger than these 4 properties, and from which they can be easily proved as corollaries. We take inspiration from [18], and find a strengthened formulation of deadlock freedom on the one hand, and strengthened memory leak freedom on the other hand. These strengthened formulations of deadlock freedom and memory leak freedom are equivalent to each other, and they imply type safety and global progress. In order to state these, we need the relation $i \; \mathsf{waiting}_\rho \; j$, which says that $i \in \mathsf{dom}(\rho)$ is waiting for $j \in \mathsf{dom}(\rho)$.

Intuitively, the meaning of $\mathsf{waiting}_\rho$ is as follows. When a barrier k gets allocated, the literal $\langle k \rangle$ appears in two threads n_1, n_2. In this case we say that $k \; \mathsf{waiting}_\rho \; n_1$ and $k \; \mathsf{waiting}_\rho \; n_2$, because the barrier is waiting until the threads want to synchronize with it. Note that this relationship is dynamic: if the barrier literal $\langle k \rangle$ is transferred to another thread, then the barrier is waiting for that new thread to synchronize with it. Whenever a thread n starts trying to synchronize with k by calling $\langle k \rangle \; v$ for some value v, then the waiting relationship flips: we now say that the thread is waiting for the barrier, *i.e.,* that $n \; \mathsf{waiting}_\rho \; k$. Formally:

▶ **Definition 3.** *We have* $i \; \mathsf{waiting}_\rho \; j$ *if either:*
1. $\rho(i) = \mathsf{Barrier}$ *and* $\rho(j) = \mathsf{Thread}(e)$ *and* $\langle i \rangle \in e$*, but* $e \neq K[\; \langle i \rangle \;\; v \,]$*, or*
2. $\rho(i) = \mathsf{Thread}(e)$ *and* $\rho(j) = \mathsf{Barrier}$*, and* $e = K[\; \langle j \rangle \;\; v \,]$

Using this notion, we can define what a partial deadlock/leak is. Intuitively, a partial deadlock is a situation in which there is some subset of the threads that are all waiting for each other. Because our notion of waiting also incorporates barriers, we generalize this to say that a *partial deadlock/leak* is a situation in which there is some subset of the threads and barriers that are all waiting for each other. Formally:

▶ **Definition 4** (Partial deadlock/leak). *Given a configuration* ρ*, a non-empty subset* $S \subseteq \mathsf{dom}(\rho)$ *is in a partial deadlock/leak if these two conditions hold:*
1. *No* $i \in S$ *can step, i.e., for all* $i \in S$*,* $\neg \exists \rho'. \, \rho \overset{i}{\leadsto} \rho'$
2. *If* $i \in S$ *and* $i \; \mathsf{waiting}_\rho \; j$ *then* $j \in S$

This notion also incorporates memory leaks: if there is some barrier that is not referenced by a thread, then the singleton set of that barrier is a partial deadlock/leak. Similarly, a single thread that is not synchronizing on a barrier, is considered to be in a singleton deadlock if it cannot step. This way, the notion of partial deadlock incorporates type safety.

▶ **Definition 5** (Partial deadlock/leak freedom). *A configuration ρ is deadlock/leak free if no $S \subseteq \mathsf{dom}(\rho)$ is in a partial deadlock/leak.*

We also strengthen the standard notion of memory leak freedom, namely reachability, to incorporate aspects of deadlock freedom.

▶ **Definition 6** (Reachability). *We inductively define the threads and barriers that are* reachable *in ρ: $j_0 \in N$ is reachable in ρ if there is some sequence $j_1, j_2, ..., j_k$ (with $k \geq 0$) such that j_0 $\mathsf{waiting}_\rho$ j_1, and j_1 $\mathsf{waiting}_\rho$ j_2, ..., and j_{k-1} $\mathsf{waiting}_\rho$ j_k, and finally j_k can step in ρ, i.e., $\exists \rho'.\, \rho \overset{j_k}{\rightsquigarrow} \rho'$.*

Intuitively, an element $j_0 \in \mathbb{N}$ is reachable if j_0 can itself step or has a transitive waiting dependency on some j_k that can step. This notion is stronger than the usual notion of reachability, which considers objects to be reachable even if they are only reachable from threads that are blocked.

▶ **Definition 7** (Full reachability). *A configuration ρ is* fully reachable *if all $i \in \mathsf{dom}(\rho)$ are reachable in ρ.*

As in [18], our strengthened formulations of deadlock freedom and full reachability are equivalent for $\bar{\lambda}$:

▶ **Theorem 8.** *A configuration ρ is deadlock/leak free if and only if it is fully reachable.*

Furthermore, these notions imply global progress and type safety:

▶ **Definition 9.** *A configuration ρ has the progress property if $\rho = \emptyset$ or $\exists \rho', i.\, \rho \overset{i}{\rightsquigarrow} \rho'$.*

▶ **Definition 10.** *A configuration ρ is safe if for all $i \in \mathsf{dom}(\rho)$, either $\exists \rho', i.\, \rho \overset{i}{\rightsquigarrow} \rho'$, or $\exists j.\, i$ $\mathsf{waiting}_\rho$ j.*

▶ **Theorem 11.** *If a configuration ρ is deadlock/leak free (or equivalently, fully reachable), then ρ has the progress and safety properties.*

Our main theorem is thus that configurations that arise from well-typed programs are fully reachable and deadlock free:

▶ **Theorem 12.** *If $\emptyset \vdash e : \mathbf{1}$ and $\{0 \mapsto \mathsf{Thread}(e)\} \rightsquigarrow^* \rho'$, then ρ' is fully reachable and deadlock/leak free.*

The proof of the reachability half of Theorem 12 proceeds similarly to the proof of global progress (Theorem 2). The difference between the proofs is that the proof of reachability needs to explicitly keep track of the reason why each object in the configuration is reachable, whereas global progress only needs to find one of the "roots" of the reachability relation (*i.e.,* threads that can step).

Theorem 8 can then be used to obtain the deadlock freedom side of Theorem 12. The idea of the proof of Theorem 8 is that the set of all unreachable objects forms a deadlock, if the set is non-empty.

6 Extending $\lambda\!\!\!\lambda$ with unrestricted and recursive types

We add unrestricted types and recursive types as extensions. These can be omitted for a minimalistic language, but they enable us to do ordinary functional programming and recursive sessions in ƛ, bringing it closer to a realistic language in terms of features and expressiveness. The extended set of types is:

$$\tau \in \mathit{Type} ::= \mathbf{0} \mid \mathbf{1} \mid \tau \times \tau \mid \tau + \tau \mid \tau \multimap \tau \mid \tau \to \tau \mid \mu a.\tau \mid a$$

An equi-recursive interpretation of $\mu x.\tau$ avoids explicit (un)fold constructs [6]. Recursive types make the language Turing complete, because they allow us to define recursive functions with the Y-combinator[8], and together with sums and products can be used to encode algebraic data types. Because session types are encoded as ordinary types in ƛ, recursive sessions are automatically supported. This includes recursion through the message, as in $\mu s.\ \mathbf{?}s.\mathsf{End}$, which is encoded as $\mu a.\ \mathbf{1} \multimap (\mathbf{1} \multimap \mathbf{1}) \times a$.

Formally and in the mechanization (Section 7), we use the coinductive method of Gay, Thiemann, and Vasconcelos [12] to handle equi-recursive types. This means that we formally do not have a syntactic $\mu a.\tau$ type constructor; instead we let the language of types be *coinductively generated*. Intuitively, this means that infinite types are allowed, and a recursive type $\mu a.F(a)$ is represented as its infinite unfolding $F(F(F(\cdots)))$. We use a meta-level fixpoint (CoFixpoint in Coq) to construct infinite/circular types. By using this method we do not need an additional typing rule for unfolding recursive types, since types are already identified up to unfolding.

In order to make interesting use of recursive types, it is necessary to add *unrestricted types*, which are types for which the linearity restriction is lifted, so that they can be duplicated and discarded freely. A linear function can only be called once and hence cannot call itself, so recursive functions must have unrestricted type. Using unrestricted and recursive types, we do not need built-in recursion as we can encode recursive functions using the Y-combinator:

$$Y : ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)) \to (\tau_1 \to \tau_2)$$
$$Y \triangleq \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Here too it is apparent that since x is used twice, unrestricted as well as recursive types are required to type check the Y-combinator [18].

In particular, we add the unrestricted function type $\tau_1 \to \tau_2$ as a new type former. We can then define rules that determine which of the existing types are unrestricted, as follows:

- $\mathbf{0}, \mathbf{1}$ are unrestricted types
- $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ are unrestricted if τ_1 and τ_2 are unrestricted
- $\tau_1 \to \tau_2$ is always unrestricted, even for linear τ_1 and τ_2
- $\tau_1 \multimap \tau_2$ is always linear, even for unrestricted τ_1 and τ_2

Using unrestricted function types, we can encode the $!A$ connective from linear logic as $\mathbf{1} \to A$, and $?A$ as $A \to \mathbf{1}$.

Formally and in the mechanization, unrestricted types are handled by splitting the typing context using a 3-part relation $\mathsf{split}\ \Gamma\ \Gamma_1\ \Gamma_2$, which intuitively says that $\Gamma \equiv \Gamma_1, \Gamma_2$, where variables of unrestricted type in Γ may occur in both Γ_1 and Γ_2 (see [18] for details). We also make use of the predicate $\Gamma\ \mathsf{unr}$ to indicate that all variables in Γ must have unrestricted type. To extend the typing rules in Figure 1, we use split to split the context, and we use

[8] One could also add an explicit **letrec**.

Γ unr wherever an empty context is required in Figure 1. For example, here are the typing rules for variables and for pairs:

$$\frac{\Gamma \ \mathsf{unr}}{\Gamma, x\!:\!\tau \vdash x : \tau} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2 \qquad \mathsf{split}\ \Gamma\ \Gamma_1\ \Gamma_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

The other rules and the rules of the run-time type system are amended analogously.

Unrestricted and recursive types are purely type-level features and require no extensions to the expression language or to the operational semantics. Nevertheless, they upgrade ƛ to a Turing complete functional and concurrent language. For more details, we refer the interested reader to the mechanization (Section 7).

7 Mechanization

All our theorems have been mechanized in Coq. We use the connectivity graph library of [18] in our mechanization. The mechanization is built-up as follows:

- Language definition: expressions, static type system, and operational semantics. Our mechanization includes the extensions with unrestricted and recursive types.
- A run-time type system that extends the static type system to barrier literals $\langle k \rangle$. The run-time type system is expressed in separation logic.
- A configuration well-formedness invariant, stating that the configuration remains well-typed, and the connectivity between threads and barriers remains acyclic.
- Proof that well-formedness is maintained by the operational semantics.
- Proof that well-formed configurations have the *full-reachability* property.
- Proofs that full-reachability is equivalent to deadlock/leak freedom, and that they imply type safety and global progress.
- The definition of the encoding of session types in ƛ, and proofs that the usual session typing rules are admissible.
- A definition of GV and its operational semantics, and the translation into ƛ, with a proof that the GV semantics can be simulated with the ƛ semantics. A lock-step simulation is obtained by inserting extra no-op steps in the GV semantics wherever ƛ does an administrative β-reduction.

Whereas the mechanized deadlock freedom proof for GV's session types by [18] consists of 2139 lines of Coq definitions and proofs (excluding [18]'s graph library), our mechanization of ƛ and its deadlock freedom is only 1229 lines. The encoding of GV's session types into ƛ together with the proofs of admissibility of the typing rules is 249 lines, and the operational simulation result is 309 lines.

Although the ƛ mechanization relies on connectivity graphs [18], the techniques presented there were not immediately sufficient for proving deadlock freedom of ƛ. The difficulty lies in ƛ's sync step in the operational semantics, which exchanges resources between two vertices that are not directly adjacent in the graph. This is not supported as an operation by [18], so we instead want to separate it into multiple smaller graph transformations. Unfortunately, the intermediate states do not satisfy the configuration invariant. The solution to this was to add extra "ghost state" to the labels on the edges of the graph, which keeps track of which sub-step of the decomposed graph transformation the connected vertices are in. As part of future work, it would be interesting to investigate whether this technique can be used more generally for composing graph transformations on the separation logic level, when the intermediate states do not satisfy the invariant.

The current version of the mechanization can be found on GitHub [17].

8 Related work

Session types were originally described by Honda [15], and later by Honda, Vasconcelos, and Kubo [16]. Gay and Vasconcelos [13] embedded session types in a linear λ-calculus. Whereas Gay and Vasconcelos' calculus [13] was not yet deadlock free, Wadler's subsequent GV [31] and its derivatives [21, 23, 24, 11, 10] were.

Wadler also described the relation of GV to classical processes (CP) [31], giving a kind of Curry-Howard correspondence between session types and classical linear logic. For intuitionistic linear logic, such a correspondence had earlier been described by Caires and Pfenning [4].

Lindley and Morris [22, 23] give a CPS encoding of GV. The target of the CPS encoding is linear λ-calculus *without* fork, which is even more minimal than $\bar{\lambda}$. The key difference between the CPS encoding and our encoding into $\bar{\lambda}$ is that the CPS encoding is *global* (*i.e.,* a whole-program transformation), whereas the encoding of sessions into $\bar{\lambda}$ is *local*, which is made possible by the built-in concurrency of $\bar{\lambda}$. In other words, in contrast to the CPS encoding, the encoding of channel operations in $\bar{\lambda}$ can be viewed as syntactic abbreviations or macros, satisfying Felleisen's expressiveness criterion [9].

When session types are added to the syntax of standard π-calculus they give rise to additional separate syntactic categories, which leads to a duplication of effort in the theory. Kobayashi showed that session types are encodable into standard π-types [19]. Dardha, Giachino, and Sangiorgi formalize and extend Kobayashi's approach [7, 8]. The encoding makes use of the fact that the π-calculus semantics has communication in the form of π-channels, and can thus encode session communication into π-communication. The encoding of multi-step sessions into single-shot π-channels sends a continuation along, so that the communication can continue. $\bar{\lambda}$'s encoding of session types takes inspiration from this work, and also sends along continuations on which the communication can continue. On the other hand, $\bar{\lambda}$ starts with λ-calculus, which does not have any concurrency or communication. We therefore *add* concurrency and communication to linear λ-calculus in the form of fork. Unlike the π-calculus' channel communication, which is one-way (like an individual step of a session), $\bar{\lambda}$'s barrier communication atomically *exchanges* two values, so that barriers may be given linear function type $A \multimap B$. This lets $\bar{\lambda}$ get away with not adding any new type formers to linear λ-calculus, by reusing the quintessential λ-calculus type (the function type) for its communication primitive.

Single-shot synchronization primitives have also been used in the implementation of a session-typed channel libraries, for instance by Scalas and Yoshida [30], Padovani [27], and Kokke and Dardha [20].

More distantly, Arslanagic, Pérez, and Voogd developed *minimal session types* [2], which decompose multi-step session types into single-step "minimal" session types of the form $!\tau.\mathsf{End}$ and $?\tau.\mathsf{End}$ in a π-calculus. Whereas $\bar{\lambda}$ and the preceding approaches [19, 7, 8] encode sequencing by nesting payload types, minimal session types "slice" the n actions of a session s into indexed names $s_1, \ldots, s_n$, each having a minimal session type. Correct sequencing is arranged on the process level with additional synchronizations, using Parrow's decomposition of processes into trios [28].

Niehren, Schwinghammer and Smolka developed a concurrent λ-calculus with futures [26]. Futures are akin to mutable variables that can only be assigned once. If a future has not been assigned a value yet, then attempting to read its value will block until a value becomes available. In addition to a non-linear type system which allows run-time errors due to multiple assignments to the same future, the authors also present a linear type system

that ensures that futures are not assigned twice. The authors are able to build channels on top of futures by starting with the ordinary linked list data type, and making the tail of the list a future, thus making the list open-ended. Unlike session-typed channels, which follow a protocol and can send values of different types at different points in the protocol, their channels always communicate values of the same type. Besides the difference between futures (which are unidirectional) and ƛ's barriers (which are bidirectional), another difference is that deadlock freedom is not guaranteed for all well-typed programs.

Aschieri, Ciabattoni and Genco give a Curry-Howard correspondence for Gödel Logic [3], which is intuitionistic logic extended with the axiom $(A \to B) \lor (B \to A)$[9]. This is a classical axiom that is implied by, but strictly weaker than the law of the excluded middle, and Gödel Logic thus provides an intermediate between intuitionistic and classical logic. The idea for the Curry-Howard interpretation of the axiom is that two copies of the continuation can be run in parallel, and exchange their evidence for A and B if both sides try to apply the implication obtained from the axiom. An important difference with ƛ is that Gödel Logic is based on intuitionistic logic (corresponding to the ordinary simply typed lambda calculus), whereas ƛ is based on the linear simply typed lambda calculus, and strongly relies on linearity for type safety and deadlock freedom. It would be interesting to investigate whether a connection between ƛ and Gödel Logic can be established, but a naive attempt at interpreting the axiom as $(A \to B) + (B \to A)$ in ƛ appears bound to fail, because in a linear setting the continuation/context cannot be duplicated without breaking the meta-theoretical properties.

We base our mechanization on the *connectivity graph* approach of Jacobs, Balzer, and Krebbers [18], and we use their library to reason about graphs in Coq. This is related to the graphical approach of Carbone [5], the proof method of Lindley and Morris [21], and to the abstract process structures of Fowler, Kokke, Dardha, Lindley, and Morris [10].

More distantly, ƛ is inspired by minimal languages such as MiniJava [29], MiniML [25], the DOT calculus [1], and others.

9 Concluding remarks

We have investigated ƛ, a minimal linear lambda calculus extended with **fork** to make it concurrent. We have seen that channel operations and linear session types can be encoded in ƛ, and we have shown that the resulting semantics for the channel operations simulates the GV semantics.

The metatheory of ƛ, including strong deadlock freedom, has been mechanized in Coq. Because of ƛ's minimality, the proofs are simpler and shorter than earlier mechanized proofs for session types. I hope you enjoyed this approach to distilling session types into a simple core, and hope that ƛ may serve as a minimal basis upon which future work may build.

References

1. Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. *The Essence of Dependent Object Types*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.
2. Alen Arslanagic, Jorge A. Pérez, and Erik Voogd. Minimal Session Types (Pearl). In *ECOOP 2019*, 2019. doi:10.4230/LIPIcs.ECOOP.2019.23.

[9] Thanks to Dan Frumin for pointing out this connection.

3 Federico Aschieri, Agata Ciabattoni, and Francesco A. Genco. Gödel logic: From natural deduction to parallel computation. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005076.

4 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236, 2010. doi:10.1007/978-3-642-15375-4_16.

5 Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *ICE*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

6 Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI*, pages 50–63. ACM, 1999. doi:10.1145/301618.301641.

7 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, 2012. doi:10.1145/2370776.2370794.

8 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

9 Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35–75, 1991. doi:10.1016/0167-6423(91)90036-W.

10 Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In *CONCUR*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.36.

11 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *PACMPL*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.

12 Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final cut. In *PLACES*, volume 314 of *EPTCS*, pages 23–33, 2020. doi:10.4204/EPTCS.314.3.

13 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

14 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010. doi:10.1016/j.ic.2010.05.002.

15 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523, 1993. doi:10.1007/3-540-57208-2_35.

16 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138, 1998. doi:10.1007/BFb0053567.

17 Jules Jacobs. Coq mechanization of lambda-barrier, 2021. The most recent version is at https://github.com/julesjacobs/cgraphs. doi:https://zenodo.org/record/6560443.

18 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. doi:10.1145/3498662.

19 Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453, 2002. doi:10.1007/978-3-540-40007-3_26.

20 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021*, Haskell 2021, 2021. doi:10.1145/3471874.3472979.

21 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOP*, volume 9032 of *LNCS*, pages 560–584, 2015. doi:10.1007/978-3-662-46669-8_23.

22 Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *Haskell Symposium*, pages 133–145, 2016. doi:10.1145/2976002.2976018.

23 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *ICFP*, pages 434–447, 2016. doi:10.1145/2951913.2951921.

24 Sam Lindley and J. Garrett Morris. Lightweight functional session types. In *Behavioural Types: from Theory to Tools*. River Publishers, 2017.

25 Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ml from higher-order logic. *SIGPLAN Not.*, pages 115–126, 2012. doi:10.1145/2398856.2364545.

26 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. In *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 248–263, 2005. doi:10.1007/11559306_14.

27 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017. doi:10.1017/S0956796816000289.

28 Joachim Parrow. Trios in concert. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 621–637. MIT Press, 1998.

29 Eric Roberts. An overview of minijava. *SIGCSE Bull.*, 2001. doi:10.1145/366413.364525.

30 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

31 Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012. doi:10.1145/2364527.2364568.

JavaScript Sealed Classes

Manuel Serrano
Inria/UCA, Inria Sophia Méditerranée, 2004 route des Lucioles, Sophia Antipolis, France

Abstract

In this work, we study the JavaScript *Sealed Classes*, which differ from regular classes in a few ways that allow ahead-of-time (AoT) compilers to implement them more efficiently. Sealed classes are compatible with the rest of the language so that they can be combined with all other structures, including regular classes, and can be gradually integrated into existing code bases.

We present the design of the sealed classes and study their implementation in the `hopc` AoT compiler. We present an in-depth analysis of the speed of sealed classes compared to regular classes. To do so, we assembled a new suite of benchmarks that focuses on the efficiency of the `class` implementations. On this suite, we found that sealed classes provide an average speedup of 19%. The more classes and methods programs use, the greater the speedup. For the most favorable test that uses them intensively, we measured a speedup of 56%.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Source code generation; Software and its engineering → Object oriented languages; Software and its engineering → Functional languages

Keywords and phrases JavaScript, Compiler, Dynamic Languages, Classes, Inline Caches

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.24

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.23`
Software (Source Code): `https://github.com/manuel-serrano/hop.git`

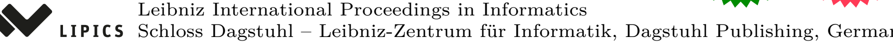
Acknowledgements My gratitude to L. Tratt for his suggestions to improve this paper, and to O. Melançon, E. Rohou, M. Feeley, and R. Findler for their comments, suggestions, and corrections.

1 Introduction

JavaScript's dynamicity makes the language very flexible and malleable. Programs are easy to adapt as their specification evolves. Data structures can be extended to meet new API requirements and extensions are easy to connect to existing code bases, etc. However, there is a flip side of this coin. First, many minor mistakes such as misspelled identifiers or omitted function arguments are usually not reported, which leads to programs that are incorrect without the programmer noticing. Second, the language is particularly difficult to implement efficiently. All fast JavaScript compilers, be they static (AoT) or dynamic (JIT), rely on heuristics and optimistic compilation to bridge the performance gap to more static languages. Consequently JavaScript programs suffer from the *performance cliff* syndrome [2, 16]: a mundane modification of the source code might dramatically affect its performance if the change, possibly very simple, defeats the compiler's heuristics.

To mitigate these problems, we explore the design and implementation of a dialect of JavaScript that extends the language with a few constructs and annotations that together allow compilers, specially AoT compilers, to generate better code. This project follows the same philosophy as JavaScript *strict mode* and *strong mode* [23]: programmers who are willing to sacrifice some dynamicity of their implementations will be rewarded with faster and more predictable performance. Sealed classes are a central component of this dialect and they are the subject of this paper.

© Manuel Serrano;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 24; pp. 24:1–24:27


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Sealed class instances and sealed class objects are class instances and class objects that are deprived of specific freedoms but that remain fully compatible with the rest of the language. All existing JavaScript programs can continue to run without any modification. Sealed classes trade some dynamicity for a more efficient implementation, in part because their design allows compilers to reuse the efficient techniques developed for class-based programming languages such as Smalltalk or Java.

In this paper we show that sealed classes are beneficial to the `hopc` [25, 26] AoT compiler. We show that they help it deliver faster and more predictable code. Although not studied in this paper, we believe that sealed classes could also benefit to JIT compilation because AoT and JIT compilers share many similar techniques for manipulating objects and properties. The main contributions of this work are:

- The characterization of lightweight constraints that allow AoT compilers to optimize the implementation of classes.
- A proof based on a complete implementation that sealed classes improve JavaScript performance of a full-fledged AoT compiler.
- An improvement to a well known technique for implementing single-inheritance class type checks.
- A benchmark suite for evaluating the performance of JavaScript classes.

The paper is organized as follows. We start in Section 2 with a brief introduction to JavaScript classes and we show why they are more difficult to implement that those of languages such as Java or C++. These difficulties motivate the introduction of sealed classes that are presented in Section 3. Their implementation is presented in Section 4. We measure their efficiency in Section 5. We present the related work in Section 6 and we conclude in Section 7.

2 Classes

Classes were added to JavaScript in version 6 [12]. They are syntactic extensions of functions [18]. They have brought a class-based programming flavor to JavaScript, without requiring any new runtime operations. Here are examples of class declarations collected from the MDN web site [19]:

```
1  class BaseClass {
2    msg = 'hello world'
3    basePublicMethod() {
4      return this.msg
5    }
6  }
7  class SubClass extends BaseClass {
8    subPublicMethod() {
9      return super.basePublicMethod()
10   }
11 }
12 const instance = new SubClass()
13 console.log(instance.subPublicMethod()) // "hello world"
```

Recently classes have been extended to support private fields. These must be declared in the class, unlike normal properties whose declarations are optional. Private properties are prefixed with '`#`'.

```
class ClassWithPrivateField {
  #priv;
  constructor() {
    this.#priv = 42;
  }
  str() {
     return `${this.#priv}`;
  }
}
class SubClass extends ClassWithPrivateField {
  #subpriv;
  constructor() {
    super();
    this.#subpriv = 23;
  }
  str() {
     return `${this.#subpriv} ${super.str()}`;
  }
}
new SubClass();
```

Private properties are only accessible from within the methods of the class that defined them. They are not accessible from within methods of subclasses or from outside the class. In our example, the `#priv` private property is only accessible from the methods of `ClassWithPrivateField`, but not from the methods of `SubClass`. Attempting to access one raises the error "*Private field must be declared in an enclosing class*".

2.1 Class Implementation

As classes are merely *special functions* [18], they are naturally implemented as a translation into functions and prototype chains. For instance, here is how the TypeScript compiler transforms the two declarations above into plain JavaScript. JavaScript compilers do an equivalent transformation internally.

```
var BaseClass = (function () {
  function BaseClass() {
    this.msg = 'hello world';
  }
  BaseClass.prototype.basePublicMethod = function () {
   return this.msg;
  };
  return BaseClass;
}());
var SubClass = (function (_super) {
  function SubClass() {
    return _super !== null && _super.apply(this, arguments) || this;
  }
  SubClass.prototype.__proto__= _super.prototype;
  SubClass.prototype.subPublicMethod = function () {
    return _super.prototype.basePublicMethod.call(this);
  };
  return SubClass;
}(BaseClass));
```

Each class is transformed into a plain JavaScript function (lines 1 and 10 in the example). The inheritance relation between a class and its superclass is implemented by chaining the prototype object of the class with the prototype object of the superclass (see at line 14 in the example). Class properties are stored in the instances themselves; methods are stored in the prototype chains. Figure 1 shows the memory layout of `BaseClass`, `SubClass`, and one instance of `SubClass`.

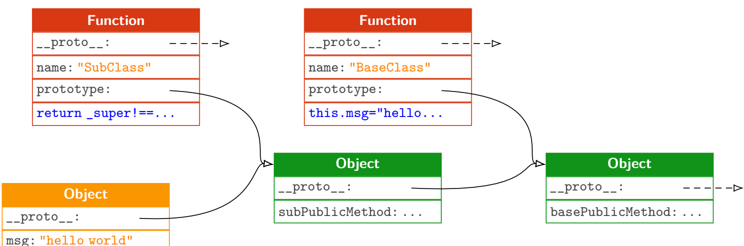


Figure 1 Two JavaScript classes and one instance.

Since a class instance is indistinguishable from a plain JavaScript object, the implementation techniques used for accessing plain objects are also those used for accessing class instances, namely inline caches [14]. We recall in the following paragraph the main principle of inline caches and hidden classes and we show their limitations when used for implementing JavaScript class accesses, re-using Serrano & Findler's presentation [28].

According to the JavaScript specification [12], accessing an object property involves the following steps:

1. convert the property name into a string S;
2. if the object owns a property S, return its value;
3. if the object has a prototype, restart at step (2) with the prototype object, return `undefined` otherwise.

The central point of the property access process is step (2). Since new properties can be added or removed at any time, checking whether an object owns a property involves looking up a key (the property name) in a dictionary (the object). Implemented literally, this protocol is several orders of magnitude slower than those of languages for which reading a structure field is a single memory read whose address is computed by adding an offset known at compile time to a base pointer.

The classic method for optimizing accesses to properties consists of associating to each object a *hidden class* and to each access a *lookup cache* [9, 3, 4]. When the property name is statically known, a very common case, a property access `obj.prop` can be implemented as follows in C (we use C to demonstrate such implementations throughout the paper):

```
1 if (obj->hclass == cache.hclass) {
2    val = obj->elements[cache.index];          // cache hit
3 } else {
4    val = cacheReadMiss(obj, "prop", &cache); // cache miss
5 }
```

When the execution environment supports dynamic code modification, the hidden class `cache.hclass` can be directly encoded as the operand of the assembly instruction implementing the test at line 1 as well as the offset `cache.index` of the property at line 2. This is where the name *inline cache* comes from.

Objects hidden classes evolve over time. One object's hidden class changes every time a property is added or removed. In the following example

```
1 const o = {};
2 o.x = 34;
3 o.y = 45;
4 delete o.x;
```

the object `o` will be successively associated with four different hidden classes: at line 1 with $h_0 = \{\}$, at line 2 with $h_1 = \{x \to 0\}$, at line 3 with $h_2 = \{x \to 0, y \to 1\}$, and at line 4 with $h_3 = \{y \to 1\}$.

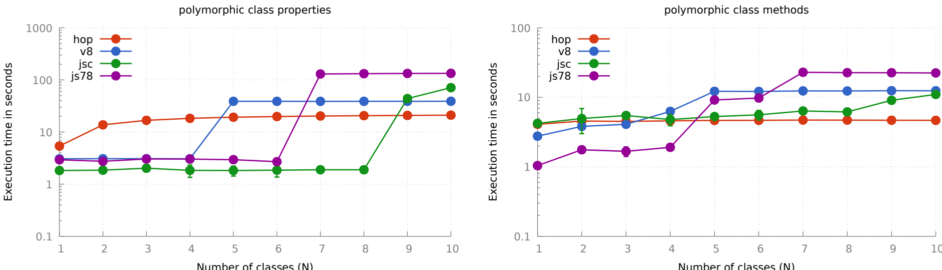


Figure 2 Impact of polymorphism on property accesses and method invocations for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), and hopc (3.6.0-pre1). Horizontal axis is the number of classes involved, aka the degree of polymorphism. The vertical axis is the execution time. Lower is better. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times and the average is reported.

Method calls are handled by similar but subtly different techniques. Class methods are stored in instance prototypes rather than in the instances themselves. Thus, for a method call, it is more efficient to store the method found in the prototype directly in the cache and to check the object proper only on method cache misses:

```
1 if (obj->hclass == cache.pclass) {
2    val = CALLN(cache.method, obj, a0, a1, ...);                 // prototype cache hit
3 } else if (obj->hclass == cache.hclass) {
4    val = CALLN(obj->elements[cache.index], obj, a0, a1, ...); // object cache hit
5 } else {
6    val = cacheMethodMiss(obj, "prop", &cache, a0, a1, ...);   // cache miss
7 }
```

This allows for an efficient method call sequence but requires a complex invalidation mechanism because when a prototype object is modified, all inline caches currently armed with that prototype method must be invalidated. This increases the unpredictability of method invocation performance.

Inline caches efficiently handle monomorphic accesses, *i.e.,* when all the objects used with one cache share the same hidden class, but they do not efficiently handle polymorphic accesses that occur when the objects accessed from a specific location have different types. For that, an enhanced technique named *Polymorphic Inline Caches* has been proposed [15]. It relies on more elaborate test sequences [20], so do properties that are not directly held by the objects themselves but by objects in the prototype chain, and properties implemented by getters and setters.

To measure the effectiveness of polymorphic caches on class implementations, we conducted an experiment following Serrano & Feeley's methodology [27]. We consider a 10-class hierarchy consisting of one base class and 9 subclasses with a maximum inheritance depth of 5. The first test repeatedly accesses a property of the base class. Each run accesses the property the same number of times, but the number of classes, N, varies. When N is 1, only one instance of a single class is used. When N is 2, two instances of two different classes are used. When N is 3, three instances of three classes are used, etc. The second experiment uses a similar principle, but adapted to method invocations. The results of these experiments are reported in Figure 2. We observe that not all systems *fall off the cliff* at the same time but, sooner or later, they all fall. Hopc is the first to fall for accessing properties when N equals 2, but it falls less deeply than the other systems with a slowdown of *only* 5×. It is followed by V8 when N is 5, then by SpiderMonkey at 7, and eventually JavaScriptCore when N is 9.

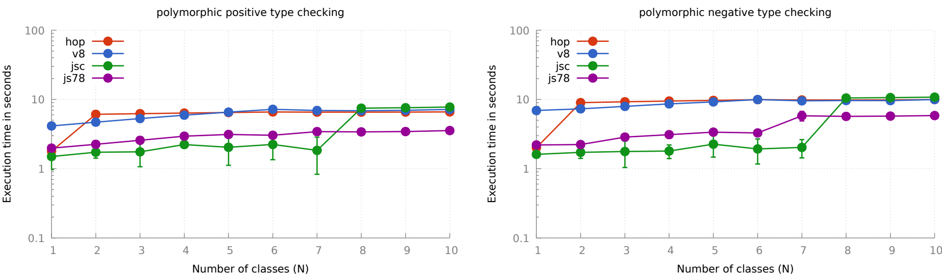


Figure 3 Impact of polymorphism on type predicates (`instanceof`) for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), and hopc (3.6.0-pre1). The figure left-hand-side shows performance when type tests succeed. The right-hand-side, shows performance when type tests fail. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.

The tuning of polymorphic inline caches have for sure been settled after careful examination of pre-class JavaScript programs, but this experiment shows that they are not well adapted for realistic applications that use class hierarchies to represent complex data structures. For instance, the hopc compiler represents its JavaScript abstract syntax tree with 52 different classes, so all the traversals that read properties from the root class would raise cache misses. Methods or properties accessed via the `super` keyword have the same limitations because they use inline cache-based implementations too [14].

The polymorphism also harms type predicates performance. This can be seen in the result of another experiment conducted with the same methodology, to evaluate the performance of the `instanceof` operator (Figure 3). Although less pronounced, we observe a similar phenomenon as with property accesses and method invocations: the performance degrades when the level of polymorphism increases.

☛ *Because instances of a class and instances of the super class are associated with different hidden classes, inline caches are not as well suited for implementing property accesses and type predicates of class-based programs as they are for regular objects.*

Implementing property accesses and type predicates efficiently is not the only difficulty of classes. Efficient instance creation is also a challenge, especially for AoT compilers, due to the dynamic nature of JavaScript class inheritance hierarchies. When declaring a class, its super class might not be known at compile time. The super-class position can be any expression, e.g., a function call or a variable reference. It is not even necessary for the super class to be another class. For instance, it may be an array or a function. This freedom has two important consequences. First, the compiler does not necessarily know the function to be invoked when compiling the `super` call (that JavaScript requires to appear before the first use of the `this` value in a constructor). This makes function inlining difficult. Second, the compiler does not know the memory size of the allocated instances. This difficulty is reinforced by instance properties that are not required to be statically declared but that can be added dynamically anywhere in the program. One proposed technique for dealing with variable allocation sizes [6] helps but the design of classes makes it complex to implement. For instance, class constructors are not even required to return the newly allocated objects, which implies extra tests if the allocation size is to be profiled.

The other difficulties come from the implementation of constructors themselves. A class is not required to declare a constructor. If it does not and if it inherits from another class, the constructor of the super class is called silently. In addition to retrieving the constructor function from the super class object and making the unknown call, a compiler faces a greater challenge: it does not necessarily know the arity of the super class constructor but,

nevertheless, all values that are passed to the constructor of the instance must be transmitted all the way up the inheritance tree, to the first declared constructor. This requires wrapping all arguments when invoking such a constructor. This has two negative impacts. First, there is cost associated with creating and initializing this wrapper. Second, it prevents constructors from receiving their arguments in registers as ordinary functions do. An alternative solution could be to use a variable arity calling convention for all constructors but it has the downside of slowing down all constructor calls.

If this meals were not already enough for the compiler, two other side dishes are also served. Inside a constructor `this` can only be used after the super constructor has been invoked, but a constructor is not required to use `this`. Resolving statically this so-called *dead zone* detection is undecidable, it is therefore not always possible to avoid generating a dynamic test that penalizes runtime efficiency. Secondly, JavaScript supports `new.target` expressions, which, inside constructors, return the classes used to create the instances. If the runtime system does not provide a way to traverse the stack for inspection purposes, `new.target` is required to be passed as an extra argument to the constructor.

☛ *The lack of static inheritance trees, the possibility of adding new properties inside or outside constructors, and the freedom to declare (or not) the constructor, make the efficient allocation and initialization of object instances challenging, especially for AoT compilers.*

Facing all these difficulties, a first attempt to trade some of the flexibility of classes for faster execution, called *strong mode*, has been proposed by the V8 development team. It is described in the following section.

2.2 JavaScript Strong Mode

The now defunct *strong mode* project from Google [23] had the dual goals to enforce static guarantees and to speed up execution. Strong mode defined a subset of JavaScript in order to forbid patterns that typically defeat compilers or that yield unpredictable performance. For instance, in strong mode, all variables had to be declared, functions have to be invoked with a number of arguments compatible with the actual number of parameters declared, and elements can not be removed from arrays.

Some restrictions were specifically designed to improve class implementation: properties could not be deleted, literal objects could not have duplicate properties, class instances were sealed after the constructor, class declarations were immutable bindings, and class constructors had to return the created object. Eventually, Google abandoned strong mode for various reasons [24]. Some of them were related to classes.

- "*Locking down classes: This was our biggest hope and the biggest failure*". For the sake of interoperability, strong mode classes were allowed to inherit from regular classes and vice versa. The team was unable to find a semantics that would work consistently and harmoniously with class inheritance and they eventually decided to give up on locking classes.
- "*ECMAScript6 classes lack property declarations*". Lacking property declarations made locking classes semantics complex because the set of properties could not be determined at creation time.
- "*ES6 performance s***!*". At that time, ECMAScript6 implementations were not yet delivering good enough performance when compared to previous JavaScript versions. Since strong mode required ECMAScript6, using strong mode resulted in a significant slowdown that was not compensated by the advantages of strong mode.
- "*Implementation complexity*". Since strong mode changes encompassed almost the entire language, its implementation was large and complex.

Since the strong mode proposal, JavaScript has evolved in several dimensions that allow us to reconsider class locking. Firstly, the class declaration itself has evolved. Optional instance properties in classes are now possible. They can even be made private, which implies that only the methods of that class itself can access them. Secondly, the language now supports modules so that it is possible to export immutable bindings, such as classes and functions.

Inspired by the strong mode endeavor and taking advantage of the latest JavaScript developments, we have designed *sealed classes*. Their design shares several constraints and restrictions with strong but relaxes others to support a gradual adoption and a smoother blending with regular classes. Thus, we believe that they can support the performance and reliability of strong mode classes while retaining most of the flexibility of regular classes. Sealed classes are presented in the next section.

3 Sealed Classes

Sealed classes are classes that are deprived of a minimal set of freedoms which, while retaining most of their flexibility, allows AoT JavaScript compilers to implement them using faster and more predictable techniques, much like class-based languages such as Smalltalk, Java, or C++. Sealed classes are syntactically similar to ordinary classes, except that their declaration is prefixed by the annotation "`// @sealed`". They are even so syntactically similar to classes that most class examples, including MDN examples, can be adapted by merely inserting the annotation on the lines preceding their declarations. All the examples of Section 2, for instance, are valid sealed classes.

Sealed classes are carefully designed so that they can be adopted gradually by programmers. For that, they are compatible with the rest of the language. They can be mixed with other data structures, including regular classes, they can be exported and imported from modules, and regular classes can inherit from sealed classes.

☛ *Sealed classes have an "erasure dynamic semantics". Ignoring the annotations of sealed classes does not change the semantics of correct programs. In consequence, any current JavaScript engine can execute correct programs using sealed classes.*

This is a central argument for adoption. This is also the reason why this paper does not include a formal semantics. The dynamic semantics of sealed classes are those of normal JavaScript classes.

Sealed classes are designed based on the observations and the results of experience presented in Section 2.1. They aim to correct the main slowness of regular classes by allowing AoT compilers to:

- use faster and more predictable techniques for implementing instance property access and method invocation in presence of polymorphism;
- use faster and more predictable type predicates;
- use faster class constructors.

Sealed classes are characterized by a set of constraints that, taken together, allow for improvements in those three dimensions. We will refer to each constraint individually when presenting their implementation, using the symbol ☆ to designate constraints that are enforced statically by the compiler, and ✪ those that require compile-time and run-time enforcements.

The key change is to allow the compiler to compute an accurate memory map of each instance, including the memory size and the offsets at which properties are stored. This is only possible if the compiler knows the whole class hierarchy. This requires that sealed

classes inherit only from other sealed classes and that the superclass hierarchy is known at compile-time. A static class hierarchy is sufficient to allow the compiler to set offsets for all properties declared in the class but it is insufficient to handle dynamically added properties on a per-instance basis, either in constructors or in the rest of the program. If this were allowed, properties declared in the class and those added dynamically would perform differently, resulting in unpredictable performance for non-expert programmers. Since sealed classes are intended to improve performance *and* predictability, we decided not to allow dynamic property (addition or removal) in sealed class instances. With this constraint, a compiler knows where all properties are stored, it can optimize their accesses, and it can raise an error if an undeclared property is accessed.

☆ **C1** *A sealed class inherits from another sealed class or from the value `null`.*
✪ **C2** *Sealed class instances are not extensible; their properties have to be declared in the class; their properties cannot be removed.*

C1 is compatible with separate compilation because the new JavaScript modules allow the compiler to know the list of imported bindings. **C1** does not prevent sealed classes from being declared and used in `eval` code because by the time a sealed class is to be evaluated, its super class already exists. In contrast, **C1** prevents an evaluated sealed class from being used as the super class of a compiled sealed class.

C2 is enforced statically by the compiler when it can track the type of an object and when it knows it is a sealed class instance. It is also enforced dynamically when either the types are unknown or when dynamic property names are used. For instance, in an expression such as "`o[`*expr*`]`", since *expr* is unknown at compile-time, a dynamic check is needed to verify that if `o` is a direct instance of a sealed class, the value of *expr* is a property of `o`.

Like sealed instances, prototypes of sealed classes are immutable and non-extensible. This is for two reasons. First, if they were extensible, the compiler could not raise errors when accessing properties that are not in the object, because the properties might be in the prototype chain. Second, if they were mutable, method invocations could not be statically resolved and would require techniques similar to inline caches, leading to unpredictability as presented in Section 2.1.

✪ **C3** *Sealed class prototypes are not extensible and they are immutable.*

C3 requires the same runtime support as **C2**.

In JavaScript, a class can be introduced in an expression or in a declaration. The constraint **C1** would prevent a sealed class expression from being the super class of another sealed class, which we believe makes sealed class expressions of little use. Therefore, for simplicity and consistency, sealed class expressions are not allowed.

☆ **C4** *Sealed classes can only be used in declarations and they are immutable.*

C1 and **C2** allow the compiler to efficiently compile property accesses when it knows that the type of the object is a sealed class, but the flexibility of plain classes makes it difficult for an AoT compiler to infer exact types. To improve the accuracy of static type analysis, two constraints are added, without which the compiler would have very little opportunity to optimize sealed class property accesses.

☆ **C5** *The constructors of sealed classes must return the freshly created instance or `void`.*
✪ **C6** *The methods of sealed classes can only be used on instances of that class or its derived classes.*

C5 guarantees that the type of a "`new SC()`" is an instance of *SC*, if *SC* is a sealed class. Note that this property does not hold if *SC* is a regular class. **C5** allows instances to be explicitly returned by the constructor for compatibility with existing code that uses JavaScript classes. **C6** guarantees that inside a method of a sealed class, the type of `this` is the sealed class. Note that this constraint does not apply to regular classes. In theory, **C6** requires type checks on the input of each method. In Section 4 we present an implementation technique that, in practice, almost completely removes them.

In the next section we discuss the implementation of sealed classes and we show how the 6 constraints that characterize them allow a compiler to exploit more efficient and more predictable techniques inspired by those of class-based programming languages. The overall performance benefit of these constraints is studied in Section 5.

4 Implementation

This section presents the main aspects of the implementation of sealed classes in the hopc AoT compiler. It shows how property access and type checking are implemented without inline caches by adapting techniques from class-based languages. It starts with the implementation of methods, and it ends with the creation of instances.

4.1 Object Representation & Properties

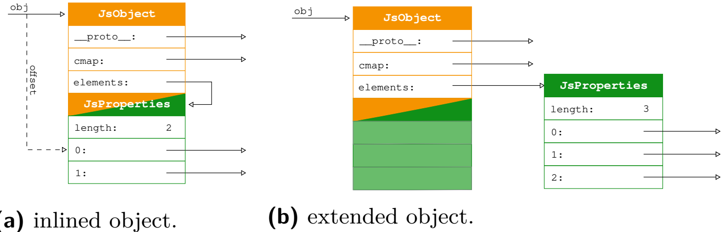
The constraints ☆ **C1**, ✪ **C2**, and ✪ **C3** allow the compiler to build complete maps of sealed class instances. Thus, for a property access "`obj.prop`", if the compiler knows that `obj` is a sealed class instance, it knows where `prop` is stored in `obj` and it no longer needs inline caches or guards, with a double benefit: dynamic checks are removed, and performance no longer depends on the degree of polymorphism (see Figure 2). When `obj` is known to be a sealed class instance, "`obj.prop`" is compiled as "`*(obj+k)`", where *k* is a constant known at compile time. ✪ **C5** and ✪ **C6** improve the precision of the type analysis. The first one gives the compiler the opportunity to track newly created sealed instances. The second one gives precise types to "`this`" in the methods of sealed classes.

Despite its apparent simplicity, this compilation scheme has subtle implications for the representation of objects. Indeed, since sealed class instances must be compatible with the rest of the runtime system, the representation of their properties must be compatible with those of regular objects, and thus with inline caches. In the remainder of this section we present the modifications applied to hopc for this purpose. First, we recall how hopc represented objects and implemented property accesses before this work [26].

Hopc uses a dual representation for objects. They are created with pre-allocated *inlined* properties. If, later, a new property is to be added, a new vector, large enough to contain the previously inlined properties and the new one, is allocated and the object representation switches from *inlined* to *extended*. The memory space previously used for the inlined properties is left unused. Figure 4 shows these two states of an object.

For each object constructor, the compiler estimates the size of the objects it creates. When an object is extended, this estimated value is incremented and the next time the constructor creates an object, it will create it larger [6]. This technique minimizes the waste of memory chunks because it reduces the number of times an object is represented as an extended object. It also maximizes the number of inlined property accesses.

Inline access is implemented as "`*(obj+cache`$_{index}$`)`", which is similar to accessing a sealed class instance property. As extended properties use one extra level of indirection their implementation requires an extra memory read: "`obj->elements[cache`$_{index}$`]`". In both



(a) inlined object. **(b)** extended object.

Figure 4 The two states of an object, inlined (4a) or extended (4b).

cases, the index where the property resides has to be discovered first. This is the purpose of hidden classes and inline caches presented in Section 2.1. In order to take advantage of inlined properties, hopc distinguishes them when checking inline caches:

```
1 if (obj->hclass == cache.iclass) {
2    val = *(obj+cache.index);                   // inline property
3 } else if (obj->hclass == cache.hclass) {
4    val = obj->elements[cache.index];           // extended property
5 } else {
6    val = cacheReadMiss(obj, "prop", &cache);   // cache miss
7 }
```

Hopc uses an encoding that makes inlined objects compatible with extended objects. When an object is inlined, its internal pointer `elements` points to the inlined chunk. When an object is extended, this pointer is updated to point to the newly allocated chunk of memory. Using a compatible representation for inlined and extended properties minimizes polymorphic inline caches, because when both encodings are used in the same location, the inline cache can be set to the extended object's hidden class, which also works for inline properties, and no cache misses will occur.

The previous inline cache sequence efficiently handles monomorphic accesses, *i.e.,* when all objects used at that source location share the same hidden class or when hidden classes distinguish between inline or extended properties, but it does not efficiently handle polymorphic accesses that occur when the objects accessed from a specific location have different types. As described in a previous study [27], to cope with polymorphism, hopc uses a long code sequence of successive tests inspired by earlier work [15]:

```
1  if (obj->hclass == cache.iclass) {
2     val = *(obj+cache.index);                                // inline property
3  } else if (obj->hclass == cache.hclass) {
4     val = obj->elements[cache.index];                        // extended property
5  } else if (obj->hclass == cache.aclass) {
6     val = obj->elements[cache.index](obj);                   // accessor property
7  } else if (obj->hclass == cache.pclass) {
8     val = cache->owner->elements[cache.index];               // prototype property
9  } else if (cache.vindex < obj->hclass->vtableLen) {
10    val = obj->elements[obj->hclass->vtable(cache.vindex)]; // polymorphic access
11 } else {
12    val = cacheReadMiss(obj, "prop", &cache);                // cache miss
13 }
```

Unfortunately this implementation of inline caches is inefficient for class-based object-oriented programming, because instances of a class and instances of a super class are associated with different hidden classes (see Section 2.1). Therefore, when accessing a property from a method of the super class, the slow path on line 10 is used. This is a major problem suffered by JavaScript classes, which is solved when the compiler knows that `obj` is a sealed class instance, because in that case the access to the property is a mere memory access without

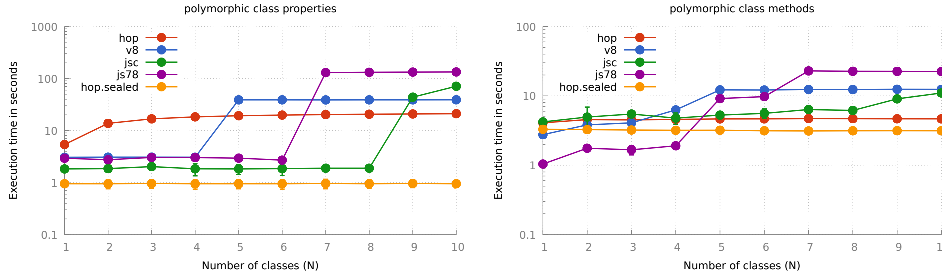
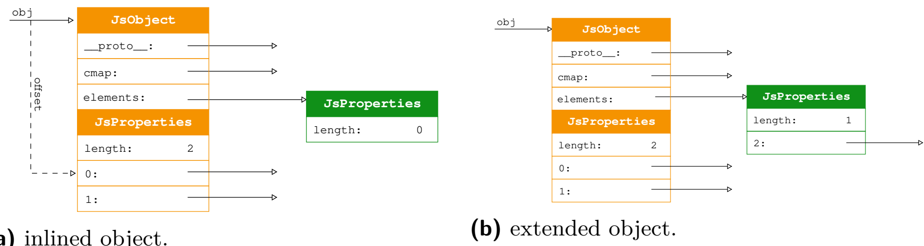


Figure 5 Impact of polymorphism on property accesses and method invocations for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), hopc (3.6.0-pre1), and hopc.sealed. Horizontal axis is the number of classes involved, aka the degree of polymorphism. The vertical axis is the execution time. Lower is better. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.



(a) inlined object.

(b) extended object.

Figure 6 The revised two states of an object, inlined (6a) or external (6b). Direct instances of sealed classes are always represented as inlined objects (6a). Other JavaScript objects, including instances of regular classes, whether their class inherits from a sealed or regular class, are represented as extended objects (6b).

any prior testing. This is the most substantial benefit of sealed classes; see the measurements in Figure 5. Sealed classes allow hopc to outperform all the other systems and allows hopc execution time to be independent of the degree of polymorphism. Note, however, that this experiment considers the best possible situation where the compiler can infer all types of all objects accessed and can then optimize all property accesses. In practice, the benefit for real programs is smaller because only a fragment a property accesses are optimized and inline caches must still be used in the majority of property accesses (see Section 5).

Despite the similarities with normal objects, sealed classes required an important change for supporting subclassing by regular classes. Class instances whose super class is a sealed class must have their properties that belong to the sealed class inlined because they need to be compatible with a sealed class representation and they also need to be extendable in order to behave as regular JavaScript objects. None of the representations of Figure 4 are compatible with this constraint. This forced us to change the hopc object implementation.

With the new encoding, objects retain their inline properties throughout their lifetime, and when they are extended, the newly allocated chunks of memory contain only additional properties (see Figure 6). This change has pros and cons. On the positive side, it reduces memory allocation and increases inlined accesses. On the negative side, it potentially increases polymorphic accesses. With the representation of Figure 4, an inlined object can be treated as an extended object because in both cases, `elements` correctly points to the property array. Then, at an access point, if both inlined and extended objects are used, the inline cache ends up being armed with the extended schema and all accesses are treated the same. With the

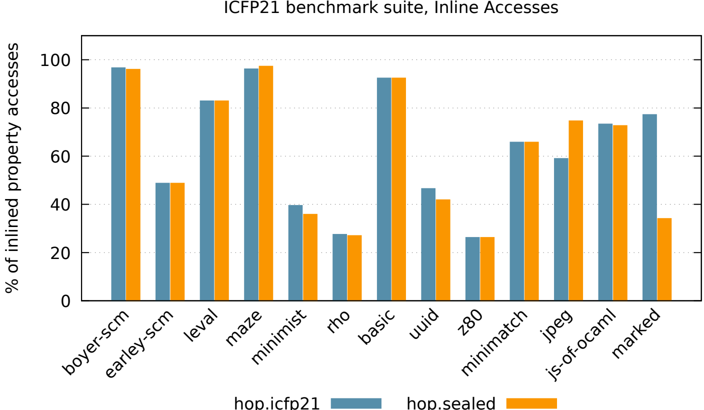


Figure 7 Percentage of inlined accesses for unmodified and modified hopc versions. Higher is better. Linear scale used.

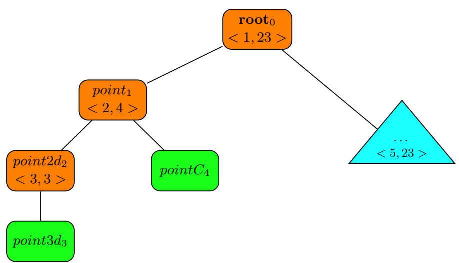


Figure 8 Cohen class numbering for single inheritance type checking.

new encoding, the inlined and extended representations are incompatible and if both are used at the same access point, they are considered as two different hidden classes, meaning it would be handled as a polymorphic access.

To measure the impact of this change, we compared the number of accesses using fast inlined property accesses for the unmodified and modified hopc versions. This experiment reuses the benchmark suite of our earlier artifact [26]. The results are presented in Figure 7. We observe that the new implementation has a marginal impact on program behaviors. The most significant impact is observed for the marked test since the new version seems to use about 40% fewer inlined properties. On a closer examination, it appears that the better behavior of the former hopc version is due to a different heuristic for allocating objects. The old version allocated them with provisional empty slots. The new version allocates them truly empty. Apart from this difference both versions behave similarly and do not show any significant performance difference.

4.2 Type Checking

Two main techniques for implementing type checking of class-based single inheritance prevail: *Cohen numbering* [7] and using an *inheritance vector*. Cohen's numbering is based on the observation that the entire class hierarchy forms a tree. The classes that are the nodes of this tree are numbered in a depth-first traversal and each class stores the class numbers of its left-most and right-most direct children (see Figure 8). To check that an instance i belongs to class $\mathcal{C}$ is done by checking that the class number of i is in the Cohen range $[\mathcal{C}_{min}..\mathcal{C}_{max}]$:

```
int isa(obj_t obj, class_t class) {
  int i = obj->class->number;
  return (i >= class->subclass_min) && (i <= class->subclass_max);
}
```

This is fast (but not as fast as the simple pointers comparison of the hidden class test) but this technique is not well suited to systems where classes are added dynamically because each addition requires renumbering the entire class hierarchy. As hopc compiles modules separately and supports dynamic loading of modules Cohen numbering is not well suited.

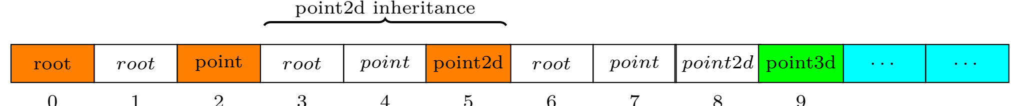
The second classical technique to implement type checking is to provide each class with a vector of its ancestors. Verifying that an instance i belongs to the class $\mathcal{C}$ is done by comparing the value of the ancestors vector at the index corresponding to the depth of $\mathcal{C}$ and $\mathcal{C}$ itself. Here is a possible implementation:

```
int isa(obj_t obj, class_t class) {
  class_t oclass = obj->class;
  return (oclass->depth <= class->depth) && (class->ancestor[class->depth] == class);
}
```

This involves five memory accesses and two integer comparisons but the dynamic addition of a class does not require any slow operation. Moreover, when the compiler statically knows the class being checked, it can compute its depth, *i.e.,* its number of ancestors, and use a faster variant:

```
int isa_depth(obj_t obj, class_t class, int depth) {
  class_t oclass = obj->class;
  return (oclass->depth <= class->depth) && (class->ancestor[depth] == class);
}
```

We use a slighty improvement of this technique by removing the range check and by removing one memory access. This required another change to the hopc object representation. Instances no longer contain a pointer to their class but rather its index. A global vector (`CLASSES`) contains all classes and their ancestors. In the vector `CLASSES`, the index of a class $\mathcal{C}$, which we notate as $\mathcal{C}_i$ contains its highest super class, *i.e.,* the root of the class hierarchy. The direct super class of $\mathcal{C}$ is stored at index $\mathcal{C}_i - 1$. Considering the class hierarchy of Figure 8 where the depth of `point2d` is 2 (it has two ancestors), if $\texttt{point2d}_i = 3$, then `CLASSES` looks like:



Checking if an instance o belongs to a class $\mathcal{C}$ is done with:

```
int isa(obj_t obj, class_t class) {
  int index = obj->class_index;
  return CLASSES[index + class->depth] == class;
}
```

This involves 4 memory accesses (as `CLASSES` is a global variable, accessing it requires a memory access).

Let us suppose that we check an instance of class $\mathcal{I}$ against a class $\mathcal{C}$. If $depth(\mathcal{C}) \leq depth(\mathcal{I})$ then comparing the value of $\texttt{CLASSES}[\mathcal{I}_i + depth(\mathcal{C})]$ and $\mathcal{C}$ implements the type check. If $depth(\mathcal{C}) > depth(\mathcal{I})$ accessing `CLASSES` will erroneously access the ancestor list of another class stored after $\mathcal{I}$. However, since $depth(\mathcal{I}) > 0$ the class that will be erroneously accessed is at an index smaller than $depth(\mathcal{C})$. Hence, it cannot be $\mathcal{C}$ because $\mathcal{C}$ is always

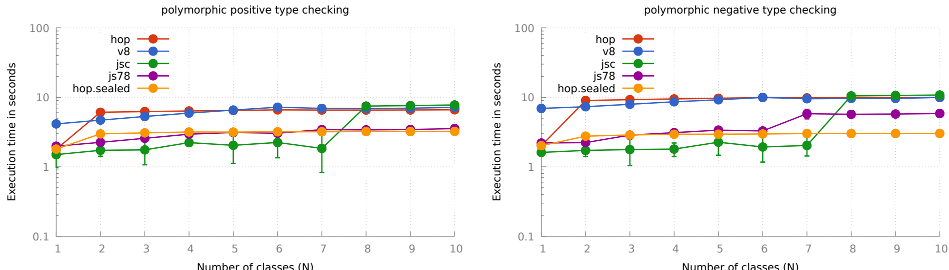


Figure 9 Impact of polymorphism on type predicates (`instanceof`) for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), and hopc (3.6.0-pre1). The figure left-hand-side shows values when type tests succeed, the right-hand-side, when they fail. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.

stored at $\mathcal{J}_i + depth(\mathcal{C})$ in `CLASSES`. Thus, boundary checks can be avoided if enough spare space is reserved at the end of `CLASSES` (this space size is the biggest inheritance depth). Finally, assuming that `class->depth` is statically known, the predicate is simplified as follows:

```
int isa_depth(obj_t obj, class_t class, int depth) {
  int index = obj->class_index;
  return CLASSES[index + depth] == class;
}
```

This routine replaces the default implementation of `instanceof` on a type check cache miss. It greatly minimizes the negative impact of polymorphism on type checking as shown in Figure 9.

4.3 Methods

Constraint **C6** requires that sealed methods be passed only instances of this class. To do this, hopc inserts dynamic checks at the beginning of each method that are compiled as:

```
class C {
   M(a0, a1, ...) {
      if (this instanceof C) {
         if (!isProxy(this)) {
            return M_unsafe(a0, a1, ...);
         } else {
            ...this:Proxy, body optimized for size...
         }
      } else {
        ...raise an error...
      }
   }
   M_unsafe(a0, a1, ...) { ...this:C, body optimized for speed... }
}
```

Inside $\mathcal{M}_{unsafe}$ the compiler knows that `this` is an instance of $\mathcal{C}$ so it optimizes property accesses, as we have already seen, but intuitively wrapping $\mathcal{M}$ also has the potential downside of requiring additional dynamic checks and extra function calls. This seldom happens.

Consider the expression "$\texttt{obj.}\mathcal{M}\texttt{(e}_0\texttt{, e}_1\texttt{, ...)}$". If the static type of `obj` is a sealed class, then the compiler rewrites the expression to "$\texttt{obj.}\mathcal{M}_{unsafe}\texttt{(e}_0\texttt{, e}_1\texttt{, ...)}$". Otherwise, it generates an inline cache sequence, for which the routine dealing with the cache miss will check whether `obj` is a sealed class. If it is, it will fill the cache with $\mathcal{M}_{unsafe}$. If it is not, it will fill it with $\mathcal{M}$, which will eventually check if it is a proxy, for which it will use a slow but compact implementation. Otherwise, it will raise an error and performance will not matter anymore. Holze et al. [15] first proposed this technique.

Constraint [✪ C3] allows the compiler to compile expressions "obj.$\mathcal{M}_{unsafe}$(e$_0$, e$_1$, ...)" without inline caches. Since prototype objects of sealed classes are immutable, the compiler statically knows all the methods of the sealed classes and groups them in a static vector. The vector of methods of a sealed class $\mathcal{C}_1$ that inherits from another class $\mathcal{C}_0$ is the concatenation of the $\mathcal{C}_0$'s inherited methods patched with overloaded methods and $\mathcal{C}_1'$s newly introduced methods. The hidden classes of sealed instances point to this vector. In pseudo C code, a sealed method invocation is then compiled as follows:

```
obj->hclass.methods[INDEX](obj, a0, a1, ...)
```

Finally a call "super.$\mathcal{M}$(e$_0$, e$_1$, ...)" inside a class $\mathcal{C}$ is implemented even more efficiently. Constraints [✪ C2], [✪ C3], and [✪ C6] allow the compiler to generate a direct call to the method $\mathcal{M}_{unsafe}$ of C's super class which, thanks to constraint [☆ C1], is statically known. Thus a call to a super method is either inlined or compiled as a direct jump.

4.4 Instance Creation

Sealed classes constraints drastically simplify instance creation. [☆ C4] allows the compiler to know which sealed class is involved in an allocation "new $\mathcal{E}$(...)". [☆ C1] and [✪ C2] allow the compiler to know the memory size of the instances to allocate at compile time. [☆ C1] allows the compiler to know the whole chain of super constructors to call when allocating an object. [✪ C2] allows the compiler to know the hidden class to associate with sealed instances, which is required for inline cache compatibility. Let us consider the following class hierarchy:

```
// @sealed
class baseclass {
   a0;
   constructor(a0) {
      this.a0 = 0;
   }
}
// @sealed
class subclass1 extends baseclass {
   a1;
}
// @sealed
class subclass2 extends subclass1 {
   a2;
   constructor(a0, a1, a2) {
      super(a0);
      this.a1 = a1;
      this.a2 = a2;
   }
}
new subclass2(0, 1, 2);
```

At line 21, the compiler knows that subclass2 instances have 3 properties and it knows the hidden class to which instances are associated, as well as their prototype. Thus, the object allocation is as follows (see Figure 6 for an explanation of the various fields):

```
JsObject *this = (subclass2)GC_malloc(sizeof(JsObject) + 3 * sizeof(JsObject *));

this->__proto__ = subclass2_PROTOTYPE; // initialize the instance prototype object
this->cmap = subclass2_HIDDENCLASS;     // initialize the instance hidden class
this->elements = 0L;                    // no extended property
this->length = 3;                       // 3 inlined properties
subclass2_CTOR(this, 0, 1, 2);          // invoke the constructor
```

The subclass2 constructor is compiled into:

```
void subclass2_CTOR(subclass2 *this, JsObject *a0, JsObject *a1, JsObject *a2) {
   {
      subclass1 *super = (subclass1 *)this; // inlining of the baseclass constructor
      *(&(super->inline0) + 0 * sizeof(JsObject *)) = a0; // set property this.a0
   }
   *(&(this->inline0) + 1 * sizeof(JsObject *)) = a1;      // set property this.a1
   *(&(this->inline0) + 2 * sizeof(JsObject *)) = a2;      // set property this.a2
}
```

In this example, since there is no constructor declared in `subclass1` and the constructor of `baseclass` is short and simple, it is inlined in the constructor of the subclasses (line 2). The compiler knows that it is not necessary to provide a data structure for `new.target` because no constructor of this class hierarchy uses it. Except for the initialization of the fields specific to JavaScript for the prototype object and the hidden class, this implementation of object allocation is similar to those of static languages.

4.5 Final Consideration

In this section we have detailed the changes made to the hopc AoT compiler to support sealed classes. These changes were necessary to enforce the constraints given in Section 3 and they all contribute to improve performance. While some changes are subtle and require fine tuning, such as the new implementation of type checking (Section 4.2), the overall size of the changes is minimal. It took less than 1,000 lines in the compiler implementation and a few hundred lines in the runtime system.

5 Sealed Class Performance Evaluation

This section contains the performance evaluation and analysis. First, it presents the benchmarks used. Next, it presents a comparison of the performance of sealed versus regular classes. Finally, it analyzes the reason for the performance differences.

5.1 Benchmarks

JavaScript classes were introduced in ECMAScript 6. Although they are more than 5 years old, they are hardly used in standard JavaScript benchmarks. Only three JetStream2 tests use them: basices2015, flightplanner, and unipoker (that we modified to use only ascii strings). We used them and other tests from different sources. First, we have created class-based versions of Octane DeltaBlue and Richards. This task was straightforward because both of these programs were originally class-based Smalltalk programs, which were translated into JavaScript by introducing prototypes to simulate class declarations and inheritance. We merely performed the reverse translation by restoring classes. When possible we used private fields instead of public fields to implement class instance properties. Similarly, we adapted the maze test used to evaluate hopc performance in a previous publication [26]. We used the two benchmarks from the StrongType study [22] that use classes (crypto-class and raytrace-class). To complete the test suite, we have ported to JavaScript the Python benchmarks of the PyPerformance test suite that use classes [29]. The similarities between the two languages allowed us to use a straightforward line-by-line translation. Two programs required extra attention. First, the bm_hexiom.py test uses the `__getitem__` so-called *magic method* to expose the values of internal instance properties. We have produced two versions of this test. The first one, hexiom where the accesses via `__getitem__` have been replaced with accesses to an array of values and the second one, hexiom-proxy, where `__getitem__` is implemented using JavaScript proxy objects. This is the only test that uses JavaScript

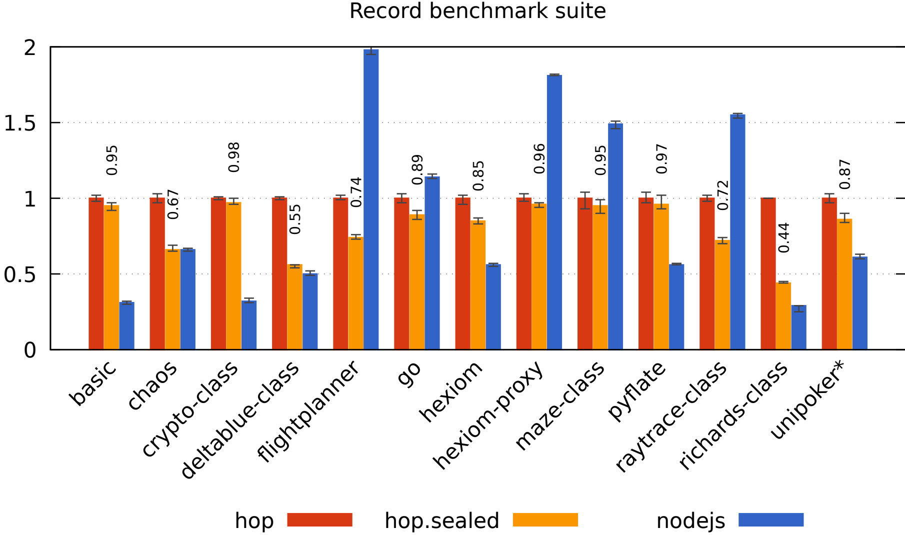


Figure 10 Sealed classes performance and V8 (9.4.146) class performance relative to Hop (3.6.0-pre1) classes performance. Lower is better. Linear scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.

proxies. The second test that required attention is bm_pyflate.py. This program uses 64bit bitwise operations which are not supported by JavaScript. We have implemented these operations using the recent JavaScript BigInt arithmetic.

5.2 Sealed Classes Performance

Figure 10 presents the overall performance evaluation. It reports the score of the unmodified hopc compiler, the score of the version supporting sealed classes (*hop.sealed*), and, for a neutral comparison, the performance of V8.[1]. We use this mainstream JIT compiler to show that on this set of benchmarks, hopc delivers a competitive performance and that the results we report in this experiment are realistic. To minimize the penalty imposed by JIT compilation we have calibrated the executions so that they last between 5 and 10 seconds.

Figure 10 shows that sealed classes are beneficial to almost all tests. The speedup can even go as high as 56% for the richards-class test. As presented in Section 4.3, the method bodies of sealed class are compiled twice: a speed-optimized version when `this` is an instance of the sealed class, and a code-size-optimized version when `this` is a proxy object. Despite this code duplication, Figure 11 shows that the speedup is not counterbalanced by a significant increase in code size. Indeed, the speed-optimized version is more compact than the implementation based on inline caches, and the difference between the two sizes is approximately the size of the compact version generated for proxy objects.

Simplifying the inline cache sequence reduces code size and accelerates execution because, by removing tests, it reduces the number of executed instructions and it minimizes the pressure on the processor's branch predictor, as we will see when studying the go test below. Figure 12 shows the percentage of simplified inline caches. Note that unipoker mostly uses arrays with so few object accesses that the measure is not relevant for that test.

[1] We observed that V8 executes faster if class properties are all replaced with instance properties added in the constructors. However, as these are an essential component of sealed classes and an official addition to JavaScript we have kept them for our evaluation.

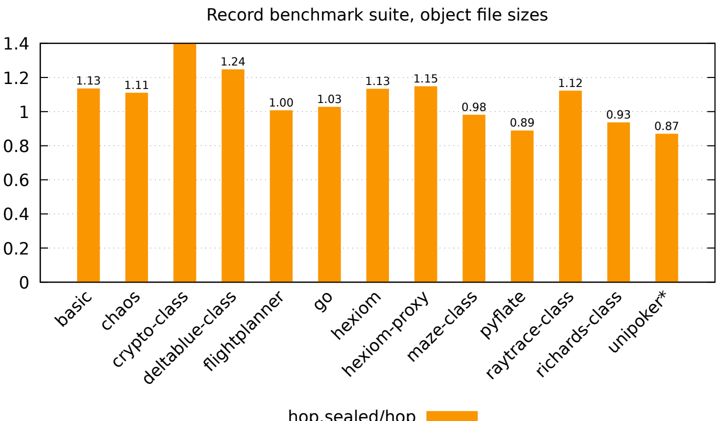


Figure 11 Sealed classes code sizes relative to Hop classes code sizes. Lower is better. Linear scale used.

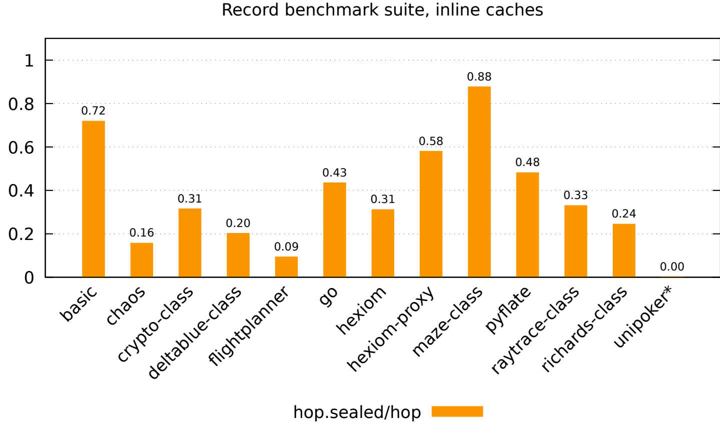


Figure 12 Sealed classes number of inline caches relative to Hop classes inline caches. Lower is better. Linear scale used.

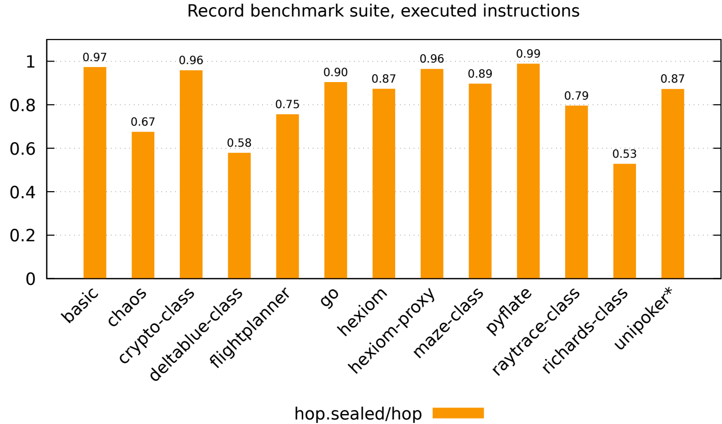


Figure 13 Comparison the number of executed instructions for classes and sealed classes. Lower is better. Linear scale used.

hop execution times (in ms)		
564.5	(24%)	GC_mark_from
230.1	(15%)	GC_add_to_black_list_normal
93.5	(10%)	hashtable-get
59.8	(08%)	&@%kwhile1286
50.4	(07%)	GC_header_cache_miss
6.0	(02%)	GC_mark_local
2.5	(02%)	GC_malloc_kind
2.2	(02%)	&eqtest
2.0	(01%)	js-map-get
1.6	(01%)	hashtable-contains?

hop.sealed times (in ms)		
447.3	(21%)	GC_mark_from
175.6	(13%)	GC_add_to_black_list_normal
120.8	(11%)	hashtable-get
92.2	(10%)	&@%kwhile1511
35.8	(06%)	GC_header_cache_miss
4.2	(02%)	GC_malloc_kind
4.2	(02%)	GC_mark_local
3.7	(02%)	js-map-get
3.6	(02%)	&eqtest
2.3	(02%)	hashtable-contains?

Figure 14 Profiling information for the basic-es2015 benchmark. Functions are reported along with the cumulative time spent executing them and the percentage of the overall execution they are responsible for. The left hand-side is the class-based hop version. The right hand-side is the modified version supporting sealed classes. The profiling values are gathered with the Linux `perf` tool.

Unexpectedly, we observe that there is no direct correlation between the static number of inline cache simplifications and the performance improvement. For instance, the pyflate test does not benefit from any significant speedup when using sealed classes, even though half of inline caches were replaced by direct property accesses without testing. Using the Linux `perf` tool [13] we measured the impact of removing the type checks of inline caches type on the number of executed instructions. This is reported in Figure 13. Again, we do not observe a strong relationship between the simplification of the generated code and the number of instructions actually executed by the processor. More generally, while all measurements (Figures 10-13) confirm the speedup of sealed classes, we cannot establish a systematic correspondence between the measured runtime reduction and the other experiments. Thus, in the following sections we conduct an in-depth analysis of some tests.

5.3 Basic.js

Basic is part of the JetStream2 suite. It implements an interpreter for the Basic ECMAScript-2015 programming language. Sealed classes do not improve basic speed. The performance difference between the two versions is below the precision of the observation. The code size is only reduced by 3% and although the number of inline caches needed to compile the program is reduced by 28%, the sealed class version does not execute fewer machine instructions than the class version.

Figure 14 shows an excerpt of the Linux `perf` profiling of the two basic versions. The numbers are the milliseconds spent in each function and the percentage of the overall execution time this corresponds to. Functions prefixed with ‘`&`’ are compiled JavaScript functions. The other functions are part of the hopc runtime system. The two compilation modes do not use the very same naming conventions for class and sealed class methods but the correspondence is fairly straightforward.

The execution time is mostly dominated by the garbage collector and by the implementation of JavaScript maps. The only client function that plays a significant role in the execution is the function named `&@kwhile1560`, as shown in the profiling report.

The test exercises JavaScript generators extensively because all the nodes of the abstract syntax tree representing the Basic source programs are encoded as generators. On an x86_64 platform, the completion of the test allocates about 6.8GB. About half of the allocated objects are arrays and the other half are generators. These allocations are unrelated to classes or sealed classes or field accesses. For instance, the hopc memory profiler reports that 23% of the overall allocations of arrays are used to prepare a function using the `apply` form.

hop execution times (in ms)	hop.sealed times (in ms)
1029.8 (32%) GC_mark_from	1103.6 (33%) GC_mark_from
442.3 (21%) GC_add_to_black_list_normal	492.4 (22%) GC_add_to_black_list_normal
90.4 (10%) GC_header_cache_miss	101.6 (10%) GC_header_cache_miss
30.5 (06%) GC_malloc_kind	25.1 (05%) &__call__%%R@Spline
13.9 (04%) &Spline.__call__	24.4 (05%) GC_malloc_kind
8.8 (03%) GC_mark_local	11.2 (03%) GC_mark_local
2.7 (02%) &GVector.linear_combination	1.7 (01%) GC_find_header
1.5 (01%) GC_find_header	1.5 (01%) GC_allochblk_nth
1.3 (01%) GC_allochblk_nth	1.0 (01%) &#transform_point@Chaosgame
0.8 (01%) &@GVector%CTOR	0.7 (01%) __pthread_mutex_trylock

Figure 15 Profiling information for the chaos benchmark.

The hopc compiler compiles generators and `yield` expressions using a CPS transformation. The `&@kwhile1560` function, which is by itself, responsible for 10% of the execution, is the compilation of a generator `while` loop. This is a regular function, not a class method, so the introduction of sealed classes does not impact its compilation. This explains why the main impact of sealed classes on basic is the code size reduction more than the execution speed.

A similar situation occurs for the go benchmark. Although the original Python version uses classes, there is no polymorphism involved and in the class JavaScript version, 99% of the accesses are optimally handled by inline caches. The sealed class version removes a significant number of these inline caches but if this reduces the size of the generated code, it does not accelerate the execution that significantly because the processor branch predictor compensates for these extra tests, as reported by the Linux `perf` report:

hop		hop.sealed	
57,510.34 msec	task-clock	56,174.36 msec	task-clock
255,988	context-switches	257,539	context-switches
9,790	cpu-migrations	10,514	cpu-migrations
4,636	page-faults	4,669	page-faults
105,678,673,722	cycles	101,498,166,407	cycles
112,477,659,094	instructions	101,682,162,341	instructions
23,867,169,583	branches	20,985,760,131	branches
396,229,049	branch-misses	392,313,687	branch-misses

We observe that the class based version executes 10% more branches than the sealed class version but the number of miss-predicted branches is almost identical for the two programs.

The same phenomenon is observed for the maze-class test. It uses classes for its data structure but it rarely uses object polymorphism. In the class-based version most accesses are efficiently handled with inline caches. As for go, the main benefit is the code size reduction.

5.4 Chaos.js

Chaos is the line-by-line transcription of the Python bm_chaos.py program. Its execution is dominated by the allocation and reclaiming of objects and boxed real numbers. The GC itself consumes about 60% of the overall execution so there is not much left for sealed classes to optimize. However, one single client function, `Spline.__call__`, is responsible for more than 10% of the overall execution, and this function allocates most of the benchmark class instances (see Figure 15). The sealed class version benefits from the faster allocation schema (see Section 4.4) to out-perform the class version.

5.5 Deltablue-class.js

Deltablue-class is a modified version of the Octane test where prototype chains are replaced with classes. The sealed class based version is significantly faster than the class based version. The acceleration comes to a large extent from the replacement of the inline caches with direct field accesses as visible in Figure 12. The more efficient polymorphic implementation of sealed classes is also an important factor of acceleration for this test. Deltablue-class defines a hierarchy of classes for representing constraints whose base class is defined as:

```
150 class Constraint {
151    strength;
152
153    constructor(strength) {
154       this.strength = strength;
155    }
```

Several classes inherit from `Constraint`, e.g., `UnaryConstraint`, which is defined as:

```
230 class UnaryConstraint extends Constraint {
231    #myOutput;
232    #satisfied;
233
234    constructor(v, strength) {
235       super(strength);
236       this.#myOutput = v;
237       this.#satisfied = false;
238       this.addConstraint();
239    }
```

When an `UnaryConstraint` instance is created, the class constructor invokes the constructor of the superclass line 235. This triggers the execution of the `Constraint` constructor and the execution of line 154. Each `Constraint`'s subclass has a dedicated hidden class, so the assignment of the `strength` property is polymorphic. hopc's inline cache profiler reports that the assignment is executed 3×10^6 times. For 1×10^6 of them, the inline cache has matched a inlined property instance but for 2×10^6, a polymorphic assignment has been executed, which involves using slower hopc vtables [27]. The replacement of classes with sealed classes enables the compiler to generate a code that always uses an inlined property assignment. Overall, the cache profiler reports that the number of polymorphic accesses and assignments drops from 56×10^6 to 8×10^6 (a 7× reduction) when switching from classes to sealed classes.

At line 235 the constructor invokes the constructor of the superclass and at line 238, it invokes the class method `addConstraint`. As `this` is the receiver of the two method invocations, the more efficient compilation of Section 4.3 contributes to accelerate this benchmark.

5.6 Flightplanner.js

Flightplanner is a benchmark of the JetStream2 suite. Sealed classes accelerate it by about 26%. The Linux `perf` report Figure 16 shows that a significant part of the acceleration is due to the simplification of the allocation of the `Leg` object that is inlined in the sealed class execution. The class execution uses slow polymorphic accesses that we observe by the significant part of the execution spent in the function `js-object-vtable-push!`. The sealed class execution only uses direct instance accesses and then never calls this function.

5.7 Raytrace-class.js

Raytrace-class is the class-based TypeScript version of the Octane as described in Richards et al. [22] from which type annotations have been erased. This is one of the few tests that uses inheritance and object polymorphism extensively, and as such, it is one of the benchmarks

hop execution times (in ms)

```
1398.8 (37%) GC_mark_from
  41.5 (06%) &@Leg%CTOR
   9.0 (03%) js-object-vtable-push!
   7.6 (03%) GC_header_cache_miss
   6.9 (03%) GC_malloc_kind
   4.6 (02%) &FlightPlan.resolveWaypoint
   4.4 (02%) GC_add_to_black_list_normal
   3.5 (02%) open-string-hashtable-get
```

hop.sealed execution times (in ms)

```
2026.8 (45%) GC_mark_from
   9.5 (03%) GC_header_cache_miss
   8.4 (03%) GC_malloc_kind
   6.2 (02%) GC_add_to_black_list_normal
   3.6 (02%) GC_allochblk_nth
   3.5 (02%) &<@resolveWaypoint%%R11757>
   2.7 (02%) string-hashtable-get
   2.1 (01%) open-string-hashtable-get
```

Figure 16 Profiling information for the Flightplanner benchmark.

hop execution times (in ms)

```
485.3 (22%) &Scheduler.schedule
310.5 (18%) &TaskControlBlock.run
127.7 (11%) &HandlerTask.run
 92.2 (10%) &TaskControlBlock.isHeldOrSusp
 52.7 (07%) &Scheduler.queue
 21.5 (05%) &TaskControlBlock.checkPriAdd
 16.1 (04%) &IdleTask.run
 13.7 (04%) &WorkerTask.run
 13.2 (04%) &DeviceTask.run
  7.4 (03%) &Scheduler.suspendCurrent
```

hop.sealed execution times (in ms)

```
621.0 (25%) &schedule%%R@Scheduler
390.9 (20%) &run%%R@TaskControlBlock
141.8 (12%) &run%%R@HandlerTask
 62.1 (08%) &queue%%R@Scheduler
 19.1 (04%) &checkPriAdd%%R@TaskControlBlock
 16.0 (04%) &release%%R@Scheduler
 15.1 (04%) &run%%R@IdleTask
 14.3 (04%) &suspendCurrent%%R@Scheduler
 12.9 (04%) &run%%R@DeviceTask
 12.5 (04%) &run%%R@WorkerTask
```

Figure 17 Profiling information for the richards-class benchmark.

that takes full benefit of sealed classes, with a speedup of 28%. Interestingly, we note that StrongScript improves the same benchmark by 22%. For this test, the benefit of the sealed class encoding seems to be similar to that of static type information. We observe a more contrasted situation for crypo-class and richards-class. StrongScript accelerates crypto-class by 6.2% but slows down richards-class by 14% while sealed classes speed them up by 3% and 56%.

5.8 Richards-class.js

Richards-class is the class-based version of the Octane test. It is the benchmark that benefits the most from sealed classes with a reduction of the execution time of about 56%. Figure 17 shows the excerpt of the Linux `perf` profiling of the two versions of the program.

The function `Scheduler.schedule` is where most of the benefit of sealed classes comes from. Its implementation is as follows:

```
1  schedule() {
2     this.#currentTcb = this.#list;
3     while (this.#currentTcb != null) {
4        if (this.#currentTcb.isHeldOrSuspended()) {
5           this.#currentTcb = this.#currentTcb.link;
6        } else {
7           this.#currentId = this.#currentTcb.id;
8           this.#currentTcb = this.#currentTcb.run();
9        }
10    }
11 }
```

This function is favorable for sealed classes because all accesses of the form `this.#`*`private-name`* are compiled as direct property accesses (see Section 4.1). In spite of this significant acceleration, the compiler's type inference is not powerful enough to infer a precise type for the expression `this.#currentTcb.link`, so hopc is not able to generate a fast sealed class method invocation and it cannot perform an efficient inlining of the methods `isHeldOrSuspended`

and `run`. The overall performance is still lagging behind the JIT V8 compiler. A better type inference, maybe one as good as those provided by TypeScript [1] or Flow [5], would significantly improve the hopc's score on this benchmark.

6 Related Work

The literature on efficient techniques for implementing classes, methods, and type checking in object-oriented programming languages is abundant and as old as these languages themselves. Implementing **Smalltalk** efficiently was a major concern [9]. Over the years, techniques have been proposed to combine static and dynamic properties [8], and techniques for implementing multiple inheritance efficiently have become necessary with languages such as **Eiffel** or **C++** [11]. In this work, we reuse these well-known techniques which we have adapted (see Section 4.2) to the JavaScript context. Our contribution is twofold. First, we propose ways to adapt classical implementation techniques designed for class-based languages so that they can cooperate with those designed to implement the dynamic features of prototype-based languages. Second, we identify some minimal restrictions that must be applied to JavaScript to allow compilers to take substantial benefit from these techniques.

Several attempts have been made to enforce stronger static guarantees and to improve the performance of JavaScript. **TypeScript** [17, 1], of course, has paved the way. It has extended JavaScript with classes before they have been integrated into the language. It has supported type annotations and static type inference in order to detect errors at compile time. A gradual type system has also been proposed to enforce runtime type correctness [21]. However, runtime acceleration is not a goal of TypeScript, in part because it compiles to plain JavaScript code and does not propose a native implementation that could take advantage of type information. This is the motivation of the **StrongType** proposal [22] which, on some benchmarks, manages to improve performance.

Retargeting the TypeScript compiler to map classes to sealed classes might be an interesting direction because the programmer interested in static error detection might also be interested in the dynamic error detection that sealed classes allow and the better performance they deliver. In the same vein, other class-based languages that compile to JavaScript, for instance **Scala.js** [10], might benefit from better performance.

Flow [5] is the Facebook JavaScript type checker. Its purpose is similar to that of TypeScript. Since Flow does not need type annotations and claims to infer more types than TypeScript, it might be interesting to try to use its type inference to statically detect classes that would obey the rules of sealed classes in order to automatically seal them for better efficiency. This is a project for future work.

Sealed classes share two motivations with Google's V8 **strong mode** endeavor [23]: enforcing static guarantees and speeding up class implementation. Strong mode defines a subset of JavaScript in order to prohibit patterns that usually defeat compilers or give unpredictable performance. Some restrictions imposed by *strong mode* are also imposed by sealed classes: properties cannot be deleted, class instances are sealed after the constructor, class declarations are immutable bindings, and class constructors must return the created object, but there are also important differences.

- Strong mode classes and regular classes were allowed to inherit from each other but for interoperability reasons, strong mode class instances and regular classes instances could not be mixed without restriction. This proved to be a show stopper for strong mode locking classes. Sealed classes take a different approach. Regular classes can inherit from sealed classes, but sealed classes cannot inherit from regular classes. This obviously restricts the use of sealed classes, but this is mandatory to improve performance while still preserving some degree of dynamicity.

- Strong mode was declared at the module level so that with a module all classes were either strong or *sloppy*. This made it difficult to adopt strong mode for existing code bases because module-level granularity proved to be too granular, making strong mode too much of an all or nothing decision. Instead, classes can be *sealed* on a declaration-by-declaration basis, one at a time, which simplifies the transition.
- The sealing of strong mode instances turned out to be impractical and incompatible with the rest of the language. Here again, sealed classes differ. A sealed class instance is sealed but an instance of a class that inherits from a sealed class is only partially sealed. Only its "*sealed class instance self*" is sealed. Its *regular class instance self* may be extended or reduced. Instances of these classes benefit from a faster implementation for their sealed part and, for their regular class part, they work like any other ordinary object.
- Sealed classes benefit from the recent JavaScript extensions that allow class field declarations, a feature not available when strong mode was proposed.
- Sealed classes are easy to implement and have minimal impact on the components of an existing system. They are very well delimited. In the implementation of the hopc compiler, apart from the modification of the object representation, they had no impact on the rest of the implementation. In total, the implementation of sealed classes represents less than 1,000 isolated lines of code in the compiler and a few hundred lines in the runtime system. In contrast, strong mode came as a whole, and strong classes could not be isolated from the rest of the proposal. The implementation was complex with ramifications in many V8 components.

7 Conclusion

Sealed classes trade a little bit of the dynamicity of JavaScript classes for faster and more predictable execution. All the benchmarks we tested benefit from sealed classes. Some benefit from a code size reduction and others benefit from speedup. Some benefit from both.

Sealed classes are compatible with the rest of the JavaScript runtime system. They can be passed to functions, returned by them, stored in data structures, and they can be used as the super classes of sealed and ordinary classes. Thus, in existing programs, sealed classes can gradually replace those classes that naturally respect the restrictions they impose. Infringements to the rules of sealed classes are detected, so that sealing classes does not present the risk of silently corrupting operational programs.

The dynamic semantics of sealed classes that do not raise errors is identical to that of regular classes. They can therefore already be used by unmodified JavaScript engines, although in this case there is no runtime acceleration. To benefit from this acceleration, we have modified the AoT hopc compiler. We have shown that the average speedup due to sealed classes is of 19% on a variety of programs using classes. This paper has detailed this implementation. It is simple and required less than 1,000 lines of new lines of code for the compiler and a few hundred lines of code for the runtime system. Sealed classes deliver better performance than regular classes *and* they are easy to implement.

References

1 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 257–281, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-662-44202-9_11.

2 Carl Friedrich Bolz. Better JIT Support for Auto-Generated Python Code. https://www.pypy.org/blog/, September 2021.

3 C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Conference Proceedings on Programming Language Design and Implementation*, PLDI ’89, New York, NY, USA, 1989. ACM. doi:10.1145/73141.74831.

4 C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA ’89, pages 49–70, USA, 1989. ACM. doi:10.1145/74878.74884.

5 Avik Chaudhuri, Basil Hosmer, and Gabriel Levi. Flow, a new static type checker for JavaScript, November 2014. URL: https://engineering.fb.com/2014/11/18/web/flow-a-new-static-type-checker-for-javascript.

6 D. Clifford, H. Payer, M. Stanton, and B. Titzer. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, New York, NY, USA, 2015. doi:10.1145/2887746.2754181.

7 N. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):626–629, 1991. doi:10.1145/115372.115297.

8 Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP ’95, pages 77–101, 1995. doi:10.1007/3-540-49538-X_5.

9 Peter L. Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’84, pages 297–302, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800017.800542.

10 Sébastien Doeraene. Cross-Platform Language Design in Scala.Js (Keynote). In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, page 1, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3241653.3266230.

11 Roland Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comput. Surv.*, 43(3):18:1–18:48, 2011. doi:10.1145/1922649.1922655.

12 ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. ECMA, 6.0 edition, June 2015.

13 Brendan Gregg. Linux systems performance. https://www.usenix.org/conference/lisa19/presentation/gregg-linux, October 2019.

14 M. Hölttä. Super fast super property access. https://v8.dev/blog/fast-super, February 2021.

15 U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP ’91, pages 21–38, UK, 1991. doi:10.1.1.126.7745.

16 B. Meurer and M. Bynens. The story of a V8 performance cliff in React. https://v8.dev/blog/react-cliff, August 2019.

17 Microsoft. TypeScript, Language Specification, version 0.9.5, November 2013.

18 Mozilla Developper Network. Classes. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes, November 2021.

19 Mozilla Developper Network. Public class fields. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Public_class_fields, July 2021.

20 F. Pizlo. Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation-in--javascriptcore, July 2020.

21 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, pages 167–180, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676971.

22 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 76–100, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2015.76.

23 Andreas Rossberg. A Strong Mode for JavaScript (Strawman proposal). URL: https://docs.google.com/document/d/1Qk0qC4s_XNCLemj42FqfsRLp49nDQMZ1y7fwf5YjaI4/view#heading=h.w5az3vf8le5k.

24 Andreas Rossberg. An update on Strong Mode, February 2016. URL: https://groups.google.com/g/strengthen-js/c/ojj3TDxbHpQ/m/5ENNAiUzEgAJ.

25 M. Serrano. Javascript aot compilation. In *14th Dynamic Language Symposium (DLS)*, Boston, USA, November 2018. doi:10.1145/3276945.3276950.

26 M. Serrano. Of AOT Compilation Performance. *Proceedings of the ACM on Programming Languages*, August 2021. doi:10.1145/3473575.

27 M. Serrano and M. Feeley. Property Caches Revisited. In *Proceedings of the 28th Compiler Construction Conference (CC'19)*, Washington, USA, February 2019. doi:10.1145/3302516.3307344.

28 M. Serrano and R. Findler. Dynamic Property Caches, a Step towards Faster JavaScripts Proxy Objects. In *Proceedings of the 29th Compiler Construction Conference (CC'20)*, San Dieo, USA, February 2020. doi:10.1145/3377555.3377888.

29 V. Stinner et al. The Python Benchmark Suite. https://github.com/python/pyperformance, 2021.

Union Types with Disjoint Switches

Baber Rehman
The University of Hong Kong, China

Xuejing Huang
The University of Hong Kong, China

Ningning Xie
University of Cambridge, UK

Bruno C. d. S. Oliveira
The University of Hong Kong, China

Abstract

Union types are nowadays a common feature in many modern programming languages. This paper investigates a formulation of union types with an elimination construct that enables case analysis (or switches) on types. The interesting aspect of this construct is that each clause must operate on *disjoint* types. By using disjoint switches, it is possible to ensure *exhaustiveness* (i.e. all possible cases are handled), and that none of the cases overlap. In turn, this means that the order of the cases does not matter and that reordering the cases has no impact on the semantics, helping with program understanding and refactoring. While implemented in the Ceylon language, disjoint switches have not been formally studied in the research literature, although a related notion of disjointness has been studied in the context of *disjoint intersection types*.

We study union types with disjoint switches formally and in a language independent way. We first present a simplified calculus, called the *union calculus* (λ_u), which includes disjoint switches and prove several results, including *type soundness* and *determinism*. The notion of disjointness in λ_u is in essence the *dual* notion of disjointness for intersection types. We then present a more feature-rich formulation of λ_u, which includes intersection types, distributive subtyping and a simple form of nominal types. This extension reveals new challenges. Those challenges require us to depart from the dual notion of disjointness for intersection types, and use a more general formulation of disjointness instead. Among other applications, we show that disjoint switches provide an alternative to certain forms of *overloading*, and that they enable a simple approach to *nullable (or optional) types*. All the results about λ_u and its extensions have been formalized in the Coq theorem prover.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Union types, switch expression, disjointness, intersection types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.25

Related Version *Full Version*: `https://github.com/baberrehman/disjoint-switches/blob/main/doc/ecoop2022-extended.pdf`

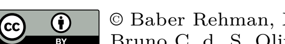
Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.17`
Software (Source Code): `https://github.com/baberrehman/disjoint-switches`

Funding This research was funded by the University of Hong Kong and Hong Kong Research Grants Council projects number 17209519, 17209520 and 17209821.

Acknowledgements We thank the anonymous reviewers for their helpful and constructive comments.

1 Introduction

Most programming languages support some mechanism to express terms with alternative types. Algol 68 [54, 55] included a form of *tagged* unions for this purpose. With tagged unions an explicit tag distinguishes between different cases in the union type. Such an approach



© Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira;
licensed under Creative Commons License CC-BY 4.0

has been adopted by functional languages, like Haskell, ML, or OCaml, which allow tagged unions (or sum types [48]), typically via either *algebraic datatypes* [15] or *variant types* [32]. Languages like C or C++ support *untagged* union types where values of the alternative types are simply stored at the same memory location. However, there is no checking of types when accessing values of such untagged types. It is up to the programmer to ensure that the proper values are accessed correctly in different contexts; otherwise the program may produce errors by accessing the value at the incorrect type.

Modern OOP languages, such as Scala 3 [44], Flow [24], TypeScript [13], and Ceylon [39], support a form of untagged union types. In such languages a union type $A \vee B$ denotes expressions which can have type A or type B. Union types have grown to be quite popular in some of these languages. A simple Google search on questions regarding union types on StackOverflow returns around 6620 results (at the time of writing), many of which arising from TypeScript programmers. Union types can be useful in many situations. For instance, union types provide an alternative to some forms of overloading and they enable an approach to *nullable types* (or explicit nulls) [34, 43].

To safely access values with union types, some form of *elimination construct* is needed. Many programming languages often employ a language construct that checks the types of the values at runtime for this purpose. Several elimination constructs for (untagged) union types have also been studied in the research literature [10, 29, 22]. Typically, such constructs take the form of a type-based case analysis expression.

A complication is that the presence of subtyping introduces the possibility of *overlapping types*. For instance, we may have a `Student` and a `Person`, where every student is a person (but not vice-versa). If we try to eliminate a union using such types we can run into situations where the type in one branch can cover a type in a different branch (for instance `Person` can cover `Student`). More generally, types can *partially overlap* and for some values two branches with such types can apply, whereas for some other values only one branch applies. Therefore, the design of such elimination constructs has to consider what to do in situations where overlapping types arise. A first possibility is to have a *non-deterministic semantics*, where any of the branches that matches can be taken. However, in practice determinism is a desirable property, so this option is not practical. A second possibility, which is commonly used for overloading, is to employ a *best-match semantics*, where we attempt to find the case with the type that best matches the value. Yet another option is to use a *first-match semantics*, which employs the order of the branches in the case. Various existing elimination constructs for unions [10, 22] employ a first-match approach. All of these three options have been explored and studied in the literature.

The Ceylon language [39] is a JVM-based language that aims to provide an alternative to Java. The type system is interesting in that it departs from existing language designs, in particular with respect to union types and method overloading. The Ceylon designers had a few different reasons for this. They wanted to have a fairly rich type system supporting, among others: *subtyping*; *generics with bounded quantification*; *union and intersection types*; and *type-inference*. The aim was to support most features available in Java, as well as a few new ones. However the Ceylon designers wanted to do this in a principled way, where all the features interacted nicely. A stumbling block towards this goal was Java-style method overloading [2]. The interaction of overloading with other features was found to be challenging. Additionally, overloaded methods with overlapping types make reasoning about the code hard for both tools and humans. Algorithms for finding the best match for an overloaded method in the presence of rich type system features (such as those in Ceylon) are challenging, and not necessarily well-studied in the existing literature. Moreover allowing overlapping

methods can make the code harder to reason for humans: without a clear knowledge of how overloading resolution works, programmers may incorrectly assume that a different overloaded method is invoked. Or worse, overloading can happen silently, by simply reusing the same name for a new method. These problems can lead to subtle bugs. For these reasons, the Ceylon designers decided not to support Java-style method overloading.

To counter the absence of overloading, the Ceylon designers turned to union types instead, but in a way that differs from existing approaches. Ceylon includes a type-based *switch construct* where all the cases must be *disjoint*. If two types are found to be overlapping, then the program is statically rejected. Many common cases of method overloading, which are clearly not ambiguous, can be modelled using union types and disjoint switches. By using an approach based on disjointness, some use cases for overloading that involve Java-style overloading with overlapping types are forbidden. However, programmers can still resort to creating non-overloaded methods in such a case, which arguably results in code easier to reason about. Disjointness ensures that it is always clear which implementation is selected for an "overloaded" method, and only in such cases overloading is allowed[1]. In the switch construct, the order of the cases does not matter and reordering the cases has no impact on the semantics, which can also aid program understanding and refactoring. Finally, from the language design point of view, it would be strange to support two mechanisms (method overloading and union types), which greatly overlap in terms of functionality.

While implemented in the Ceylon language, disjoint switches have not been studied formally. To our knowledge, the work by Muehlboeck and Tate [42] is the only work where Ceylon's switch construct and disjointness are mentioned. However, their focus is on algorithmic formulations of distributive subtyping with unions and intersection types. No semantics of the switch construct is given. Disjointness is informally defined in various sentences in the Ceylon documentation. It involves a set of 14 rules described in English [1]. Some of the rules are relatively generic, while others are quite language specific. Interestingly, a notion of disjointness has already been studied in the literature for intersection types [45]. That line of work studies calculi with intersection types and a *merge operator* [49]. Disjointness is used to prevent ambiguity in merges, which can create values with types such as $\mathsf{Int} \wedge \mathsf{Bool}$. Only values with disjoint types can be used in a merge.

In this paper, we study union types with disjoint switches formally and in a language independent way. We present the *union calculus* (λ_u), which includes disjoint switches and union types. The notion of disjointness in λ_u is interesting in the sense that it is the dual notion of disjointness for intersection types. We prove several results, including *type soundness*, *determinism* and the *soundness* and *completeness* of algorithmic formulations of disjointness. We also study several extensions of λ_u. In particular, the first extension (discussed in Section 4) adds intersection types, nominal types and distributive subtyping to λ_u. It turns out such extension is non-trivial, as it reveals a challenge that arises for disjointness when combining union and intersection types: the dual notion of disjointness borrowed from disjoint intersection types no longer works, and we must employ a novel, more general, notion instead. Such change also has an impact on the algorithmic formulation of disjointness, which must change as well. We also study two other extensions for parametric polymorphism and a subtyping rule for a class of empty types in the extended version of this paper. We prove that all the extensions retain the original properties of λ_u. Furthermore, for our subtyping relation in Section 4 we give a *sound*, *complete* and *decidable* algorithmic formulation by extending the algorithmic formulation employing *splittable types* by Huang and Oliveira [36].

[1] Ceylon does allow dynamic type tests, which in combination with switches can simulate some overlapping.

To illustrate the applications of disjoint switches, we show that they provide an alternative to certain forms of *overloading*, and they enable a simple approach to *nullable (or optional) types*. All the results about λ_u and its extensions have been formalized in the Coq theorem prover. In summary, the contributions of this paper are:

- **The λ_u calculus:** We present a simple calculus with union types, nullable types and a disjoint switch construct. We then present a richer extension of λ_u with intersection types, distributive subtyping and nominal types. In addition, in the extended version of the paper, we study extensions with parametric polymorphism and a subtyping rule to detect empty types. All calculi and extensions are type sound and deterministic.
- **Sound, complete and decidable formulations of disjointness and subtyping:** We present two formulations of disjointness, which are general and language independent. The second formulation is novel and more general, and can be used in a calculus that includes intersection types as well. We also extend a previous subtyping relation [36] to include nominal types. For both disjointness and subtyping we show that the specifications are sound, complete and decidable and present the corresponding algorithmic formulations.
- **Mechanical formalization:** The results about λ_u and its extensions have been formalized in the Coq theorem prover and can be found in the supplementary materials, together with an extended version of the paper.

2 Overview

This section provides some background on union types and some common approaches to eliminate union types. Then it describes the Ceylon approach to union types, and discusses a few applications of union types. Finally, it presents the key ideas and challenges in our work.

2.1 Tagged Union Types

We start with a brief introduction to union types. An expression has a union type $A \vee B$, if it can be considered to have either type A *or* type B. Many systems model *tagged union types* (also called *sum types* or *variants types*), where explicit *tags* are used to construct terms with union types, as in languages with algebraic datatypes [15] or (polymorphic) variants [32]. In their basic form, there are two introduction forms: $\mathsf{inj}_1 : A \to A \vee B$ turns the type of an expression from A into $A \vee B$; and $\mathsf{inj}_2 : B \to A \vee B$ turns the type of an expressions from B into $A \vee B$. Using tagged union types, we can implement a safe integer division function, as[2]:

```
String | Int safediv (x : Int) (y : Int) =
  if (y == 0) then inj1 "Divided by zero" else inj2 (x / y)  // uses tags
```

Here the intention is to have a safe (integer) division operation that detects division by zero errors, and requires clients of this function to handle such errors. The return type `String | Int` denotes that the function can either return an error message (a string), or an integer, when division is performed without errors.

[2] Throughout this paper, we write union types as `A | B` in code, since this is widely adopted in programming languages (e.g., Ceylon, Scala, and TypeScript), and as $A \vee B$ in the formal calculi, which is more frequently used in the literature.

Elimination form for tagged union types. Tagged union types are eliminated by some form of case analysis. For consistency with the rest of the paper, we use a syntactic form with `switch` expressions for such case analysis. For example, the following program `tostring` has different behaviors depending on the tag of `x`, where `show` takes an `Int` and returns back its string representation.

```
String tostring (x: String | Int) = switch (x)
                                      inj1 str -> str
                                      inj2 num -> show num
```

2.2 Type-directed Elimination forms for Union Types

While tags are useful to make it explicit which type a value belongs to, they also add clutter in the programs. On the other hand, in systems with subtyping for union types [29, 47, 42], explicit tags are replaced by implicit coercions represented by the two subtyping rules $A <: A \vee B$ and $B <: A \vee B$. In this paper we refer to union types where the explicit tags are replaced by implicit coercions as *untagged union types*, or simply *union types*. In those systems, a term of type A or B can be directly used as if it had type $A \vee B$, and thus we can write safe division as:

```
String | Int safediv2 (x : Int) (y : Int) =
  if (y == 0) then "Divided by zero" else (x / y)       // no tags!
```

However, now the elimination form of union types cannot rely on explicit tags anymore, and different systems implement elimination forms differently. The most common alternative is to employ types in the elimination form. We review type-directed union elimination next.

Type-directed elimination. Some systems [22] support *type-directed* elimination of union types. For instance, `tostring2` has different behaviors depending on the *type* of `x`.

```
String tostring2 (x: String | Int) = switch (x)
                                       (y : String) -> y
                                       (y : Int)    -> show y
```

However, compared to tag-directed elimination, extra care must be taken with type-directed elimination. In particular, while we can easily distinguish tags, ambiguity may arise when types in a union type overlap for type-directed elimination. For example, consider the type `Person | Student`, where we assume `Student` is a subtype of `Person`. With type-directed elimination, we can write:

```
Bool isstudent (x: Person | Student) = switch (x)
                                         (y : Person)  -> False
                                         (y : Student) -> True
```

Now it is unclear what happens if we apply `isstudent` to a term of type `Student`, as its type matches both branches. In some calculi [29], the choice is not determined in the semantics, in the sense that either branch can be chosen. This leads to a non-deterministic semantics. In some other languages or calculi [22], branches are inspected from top to bottom, and the first one that matches the type gets chosen. However, in those systems, as `Person` is a supertype of `Student`, the first branch subsumes the second one and will always get chosen, and so the second branch will never get evaluated! This may be unintentional, and similar programs being accepted can lead to subtle bugs. Even if a warning is given to alert programmers that a case can never be executed, there are other situations where two cases overlap, but neither case subsumes the other. For instance we could have `Student` and `Worker` as subtypes of `Person`.

For a person that is both a student and a worker, a switch statement that discriminates between workers and students could potentially choose either branch. However for persons that are only students or only workers, only one branch can be chosen.

Best-match and overloading. Some languages support an alternative to typed-based union elimination via method overloading. Such form is used in, for example, Java [33] and Julia [57]. In Java, we can encode `isstudent2` as an overloaded method, which has different behaviors when the type of the argument differs.

```
boolean isstudent2 (Person x)  { return False; }
boolean isstudent2 (Student x) { return True; }
```

Java resolves overloading by finding and selecting, from all method implementations, the one with the *best* type signature that describes the argument. If we apply `isstudent2` to a term of type `Student`, the second implementation is chosen, as `Student` is the best type describing the argument. As we can see, such a best-match strategy eliminates the order-sensitive problem, as it always tries to find the best-match despite the order. That is, in Java the method order does not matter: in this case, we have the method for `Person` before the one for `Student`, but Java still finds the one for `Student`.

However, the best-match strategy can also be confusing, especially when the system features implicit upcasting (e.g., by subtyping). If programmers are not very familiar with how overloading resolution works, they may assume that the wrong implementation is called in their code. For instance, in Java we may write:

```
Person p = new Student();
isstudent2(p);
```

In this case Java will pick the `isstudent2` method with the argument `Person`, since Java overloading uses the *static type* (`p` has the static type `Person`) to resolve overloading. But some programmers may assume that the implementation of the method for `Student` would be chosen instead, since the person is indeed a student in this case. This can be confusing and lead to subtle bugs.

Moreover, there are other tricky situations that arise when employing a best-match strategy. For example, suppose that the type `Pegasus` is a subtype of both type `Bird` and type `Horse`. If a method `isbird` is overloaded for `Bird` and `Horse`, then which method implementation should we choose when we apply `isbird` to a term of type `Pegasus`, the one for `Bird`, or the one for `Horse`? In such case, we have an ambiguity. Things get worse when the type system includes more advanced type system features, such as generics, intersections and union types, or type-inference.

2.3 Union Types and Disjoint Switches in Ceylon

The Ceylon language [39] supports type-directed union elimination by a switch expression with branches. The following program is an example with union types using Ceylon's syntax:

```
void print(String|Integer|Float x) {
  switch (x)
    case (is String)        { print("String: ``x``"); }
    case (is Integer|Float) { print("Number: ``x``"); }
}
```

For the switch expression, Ceylon enforces static type checking with two guarantees: *exhaustiveness*, and *disjointness*. First, Ceylon ensures that all cases in a switch expression are *exhaustive*. In the above example, `x` can either be a string, an integer or a floating point

number. The types used in the cases do not have to coincide with the types of `x`. Nevertheless, the combination of all cases must be able to handle all possibilities. If the last case only dealt with `Integer` (instead of `Integer|Float`), then the program would be statically rejected, since no case deals with `Float`.

Second, Ceylon enforces that all cases in a switch expression are *disjoint*. That is, unlike the approaches described in Section 2.2, in Ceylon, it is impossible to have two branches that match with the input at the same time. For instance, if the first case used the type `String | Float` instead of `String`, the program would be rejected statically with an error. Indeed, if the program were to be accepted, then the call `print(3.0)` would be ambiguous, since there are two branches that could deal with the floating point number. Note that, since the cases in a switch cannot overlap, their order is irrelevant to the program's behavior and its evaluation result. All of the overlapping examples from the previous section will statically be rejected in similar fashion.

Union types as an alternative to overloading. One motivation for such type-directed union elimination in Ceylon is to model a form of function overloading. The following example, which is adapted from TypeScript's documentation [3], demonstrates how to define an "overloaded" function `padLeft`, which adds some padding to a string. The idea is that there can be two versions of `padLeft`: one where the second argument is a string; and the other where the second argument is an integer:

```
String space(Integer n){
  if (n==0) { return ""; }
  else      { return " "+space(n-1); }
}
String padLeft(String v, String|Integer x) {
  switch (x)
    case (is String)  { return x+v; }
    case (is Integer) { return space(x)+v; }
}
print(padLeft("?", 5)); // "     ?"
print(padLeft("World", "Hello ")); // "Hello World"
```

In `padLeft`, there are two cases of the switch construct depending on the type of `x`: the first one appends a string to the left of `v`, and the other calls function `space` to generate a string with `x` spaces, and then append that to `v`. Although statically `x` has type `String|Integer`, as a concrete value it can only be a string or an integer. As such, when values with such types are passed to the function, the corresponding branch is chosen and executed.

2.4 Nullable Types

Besides being used for overloading, union types can be used for other purposes too. *Null pointer exceptions (NPEs)* are a well-known and tricky problem in many languages. The problem arises when dereferencing a pointer with the `null` value. For instance, if we have a variable `str`, which is assigned to `null`, the the code `print(str.size)`, in a Java-like language, will raise a null pointer exception. This is because of so-called implicit nulls in Java and other popular languages. With implicit nulls, any variable of a reference type can be `null`.

An interesting application of union types in Ceylon is to encode *nullable types* (or *optional types*) [34] in a type-safe way. A similar approach to nullable types has also been recently proposed for Scala [43]. In those languages, there is a special type `Null`, which is inhabited by the `null` value. Note that `Null` differs from `Nothing` (the bottom type in Ceylon), in the sense that `Null` is inhabited while `Nothing` is not. To illustrate the subtle difference, Figure 1 presents a part of the subtyping lattice in Ceylon. `Anything`, the top type in Ceylon, is an

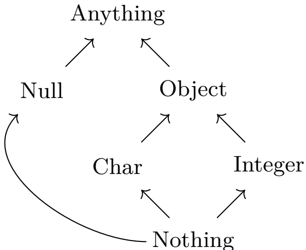


Figure 1 Ceylon's subtyping hierarchy. Note that `Null` only has `Nothing` as its subtype.

enumerated class. `Anything` is also a supertype of `Object`, which is the root of primitive types, function types, all interfaces and any user-defined class. Notably, `Null` is disjoint to `Object`, and therefore, to all user-defined classes.

In Ceylon the following code:

```
String str = null;
```

is rejected with a type error, since `null` cannot have type `String`. Instead, a type that can have the null value must be defined explicitly in Ceylon using union types:

```
String | Null str = null;
```

Now we cannot call `str.size`, as `str` may be `null`, and `size` is not defined on `null`. To get the `size` of `str`, we must first check whether `str` is `null` or not using disjoint switches:

```
String | Null str = null;
switch (str)
  case (is String) { print(str.size); }
  case (is Null)   { print(); }
```

Other uses of Union Types. Union types are also useful in many other situations. In Section 2.2 we illustrated a `safediv` operation, which can be easily encoded in Ceylon as:

```
String | Integer safediv3 (Integer x, Integer y){
  if (y==0) { return "Divided by zero"; }
  else      { return (x/y); }
}
```

The return value can be a string or an integer, with no explicit tag needed, as union types are implicitly introduced. As long as the declared return type of the function is a supertype of all possible return values, it is valid in Ceylon.

2.5 Key Ideas in Our Work

We first introduce a simplified formulation of λ_u, which formalizes the basic ideas of union types with disjoint switches similar to those in the Ceylon Language. To our best knowledge, there is yet no formalism of disjoint switches, and λ_u studies those features formally and precisely. In particular, λ_u captures the key idea for type-directed elimination of union types in its switch construct in a language independent way, and formally defines disjointness, disjointness and subtyping algorithms, and the operational semantics. The simplified formulation of λ_u is useful to compare with existing calculi with union types in the literature [41, 9, 30, 29, 47, 18]. Moreover, we study a more fully featured formulation of λ_u that includes practical extensions, such as intersection types, distributive subtyping, nullable

types and a simple form of nominal types. λ_u is proved to enjoy many desirable properties, such as type soundness, determinism and the soundness/completeness of disjointness and subtyping definitions. All the Ceylon examples in Sections 2.3 and 2.4 can be encoded in λ_u.

Disjointness. A central concept in the formulation of disjoint switches is disjointness. Our first hurdle was to come up with a suitable formal definition of disjointness. Consider the simple λ_u switch expression:

```
switch x {
  (y : String | Int)  -> 0
  (y : Int | Bool)    -> 1
}
```

Here we wish to determine whether $\mathsf{String} \vee \mathsf{Int}$ and $\mathsf{Int} \vee \mathsf{Bool}$ are disjoint or not. In other words, we wish to determine whether, for any possible (dynamic) type that x can have, it is unambiguous which branch to choose. In this case, it turns out that there is ambiguity. For instance, if x is an integer, then either branch can be chosen. Thus λ_u rejects this program with a disjointness error. In this example, the reason to reject the program is basically that $\mathsf{Int} <: \mathsf{String} \vee \mathsf{Int}$ and $\mathsf{Int} <: \mathsf{Int} \vee \mathsf{Bool}$. That is we can find a common subtype (Int) of the types in both branches. Moreover, that subtype can be inhabited by values (integer values in this case). If the only common subtypes of the types in the two branches would be types like $\bot$ (which has no inhabitants), then the switch should be safe because we would not be able to find a value for x that would trigger two branches. This observation leads to the notion of disjointness employed in the first variant λ_u in Section 3. Formally, we have:

▶ **Definition 1** ($\bot$-Disjointness). $A * B ::= \forall\ C$*, if* $C <: A$ *and* $C <: B$ *then* $\rfloor C \lfloor$

Here we use $\rfloor C \lfloor$ to denote that type C is equivalent to type $\bot$, or, bottom-like (i.e. $C <: \bot$). In either definition, Int serves as a counter-example for $\mathsf{String} \vee \mathsf{Int}$ and $\mathsf{Int} \vee \mathsf{Bool}$ to be disjoint. Thus λ_u rejects the program above with a disjointness error. It is worth noting that this first notion of disjointness is essentially dual to a definition of disjointness for intersection types in the literature in terms of top-like common supertypes [45].

Disjointness in the presence of intersection types. The variant of λ_u in Section 3 does not include intersection types. Unfortunately, the disjointness definition above does not work in the presence of intersection types. The reason is simple: with intersection types we can always find common subtypes, such as $\mathsf{Int} \wedge \mathsf{Bool}$, which are not bottom-like, and yet they have no inhabitants. That is, $\mathsf{Int} \wedge \mathsf{Bool}$ is not a subtype of $\bot$, but no value can have both type Int and type Bool. We address this issue by reformulating disjointness in terms of *ordinary types* [28], which are basically primitive types (such as integers or functions). If we can find common *ordinary* subtypes between two types, we know that they are not disjoint. Thus the disjointness definition used for formulations of λ_u with intersection types is:

▶ **Definition 2** ($\wedge$-Disjointness). $A * B ::= \nexists\ C^{\circ},\ C^{\circ} <: A$ *and* $C^{\circ} <: B$.

Note that here C° is a metavariable denoting ordinary types. Under this definition we can check that Int and Bool are disjoint, since no ordinary type is a subtype of both of these two types. This definition avoids the issue with Definition 1, which would not consider these two types disjoint. Moreover, this definition is a generalization of the previous one, and in the variant with union types only the two definitions coincide.

This new definition requires a different approach to algorithmic disjointness. Our new approach is to use the notion of *lowest ordinary subtypes*: For any given type, we calculate a finite set to represent all the possible values that can match the type. Then we can easily determine whether two types are disjoint by ensuring that the intersection of their lowest ordinary subtypes is empty.

Distributive Subtyping. In Section 4, we study λ_u with an enriched distributive subtyping relation inspired by Ceylon programming language. Distributive subtyping is more expressive than standard subtyping and adds significant complexity in the system, in particular for a formulation of algorithmic subtyping and the completeness proof of the disjointness algorithm. Nevertheless, distributive subtyping does not affect the disjointness definition and its algorithm remains the same with and without distributive subtyping. The following code snippet elaborates on the expressiveness of distributive subtyping:

```
void do (<Integer & String> | Boolean val) { /* do something */ }
```

The function `do` in above code snippet takes input value of type $(\mathsf{Int} \wedge \mathsf{String}) \vee \mathsf{Bool}$. However, we cannot pass a value of type $(\mathsf{Int} \vee \mathsf{Bool}) \wedge (\mathsf{String} \vee \mathsf{Bool})$ to the function `do`: we get a type error if we try to do that in a system with standard subtyping (without distributivity), as standard subtyping fails to identify that the value has a subtype of the expected argument type. Distributive rules enable this subtyping relation. With distributivity of unions over intersections (and vice-versa), the type $(\mathsf{Int} \vee \mathsf{Bool}) \wedge (\mathsf{String} \vee \mathsf{Bool})$ is a subtype of $(\mathsf{Int} \wedge \mathsf{String}) \vee \mathsf{Bool}$ (in particular, by rule DS-DISTOR in Figure 6). As such with distributive subtyping, the following Ceylon program type-checks:

```
variable <Integer | Boolean> & <String | Boolean> x = true; do(x);
```

Nominal Types and Other Extensions to λ_u. We also study several extensions to λ_u, including nominal types. The extension with nominal types is interesting, since nominal types are highly relevant in practice. We show a sound, complete and decidable algorithmic formulation of subtyping with nominal types by extending an approach by Huang and Oliveira [36]. We show that disjointness can also be employed in the presence of nominal types. This extension rejects ambiguous programs with overlapping nominal types in different branches of switch construct at compile time. It illustrates that disjointness is practically applicable to structural types as well as the nominal types. For example, the following program will statically be rejected in λ_u with nominal types:

```
Bool isstudent (x: Person | Student) = switch (x)
                                        (y : Person)  -> False
                                        (y : Student) -> True
```

Whereas, the following program will be accepted if we know that `Person` and `Vehicle` are disjoint:

```
Bool isvehicle (x: Person | Vehicle) = switch (x)
                                        (y : Person)  -> False
                                        (y : Vehicle) -> True
```

3 The Union Calculus λ_u

This section introduces the simplified union calculus λ_u. The distinctive feature of the λ_u calculus is a type-based switch expression with disjoint cases, which can be used to eliminate values with union types. In this first formulation of λ_u we only include the essential features

Type	A, B, C	::=	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid \mathsf{Null}$
Expr	e	::=	$x \mid i \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \mid \mathsf{null}$
Value	v	::=	$i \mid \lambda x.e \mid \mathsf{null}$
Context	Γ	::=	$\cdot \mid \Gamma, x : A$

$\boxed{A <: B}$ *(Subtyping)*

$$\frac{}{A <: \top}\ \text{S-TOP} \qquad \frac{}{\mathsf{Null} <: \mathsf{Null}}\ \text{S-NULL} \qquad \frac{}{\mathsf{Int} <: \mathsf{Int}}\ \text{S-INT}$$

$$\frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \text{S-ARROW} \qquad \frac{}{\bot <: A}\ \text{S-BOT} \qquad \frac{A <: C \qquad B <: C}{A \vee B <: C}\ \text{S-ORA}$$

$$\frac{A <: B}{A <: B \vee C}\ \text{S-ORB} \qquad \frac{A <: C}{A <: B \vee C}\ \text{S-ORC}$$

Figure 2 Syntax and subtyping for λ_u.

of a calculus with disjoint switches: union types and disjoint switches. Section 4 then presents a richer formulation of λ_u with several extensions of practical relevance. We adapt the notion of disjointness from previous work on *disjoint intersection types* [45] to λ_u, and show that λ_u is type sound and deterministic.

3.1 Syntax

Figure 2 shows the syntax for λ_u. Metavariables A, B and C range over types. Types include top ($\top$), bottom ($\bot$), integer types (Int), function types ($A \to B$), union types ($A \vee B$) and null types (Null). Metavariable e ranges over expressions. Expressions include variables (x), integers (i), lambda abstractions ($\lambda x.e$), applications ($e_1\ e_2$), a novel switch expression ($\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\}$) and the null expression. The `switch` expression evaluates a specific branch by matching the types in the cases. Note that, although the switch expression in λ_u only has two branches, a multi-branch switch can be easily encoded by employing nested switch expressions. We model the two-branch switch for keeping the formalization simple, but no expressive power is lost compared to a multi-branch switch. Metavariable v ranges over values. Values include i, $\lambda x.e$ and null expressions. Finally, a context (Γ) maps variables to their associated types.

3.2 Subtyping

The subtyping rules for λ_u are shown at the bottom of Figure 2. The rules are standard. Rule S-TOP states that all types are subtypes of the $\top$ type. Rule S-BOT states that $\bot$ type is subtype of all types. Rule S-NULL states that the Null type is a subtype of itself. Rules S-INT and S-ARROW are standard rules for integers and functions respectively. Functions are contravariant in input types and covariant in output types. Rules S-ORA, S-ORB, and S-ORC deal with the subtyping for union types. Rule S-ORA says that the union type of A and B is a subtype of another type C if both A and B are subtypes of C. Rules S-ORB and S-ORC state if a type is subtype of one of the components of a union type, then it is subtype of the union type. The subtyping relation for λ_u is reflexive and transitive.

3.3 Disjointness

The motivation for a definition of disjointness based on bottom-like types is basically that in disjoint switches, the selection of branches can be viewed as a type-safe downcast. For instance, recall the example in Section 2.5:

```
switch x {
   (y : String | Int)  -> 0
   (y : Int | Bool)    -> 1
}
```

Here `x` may have type `Int | String | Bool` and the two branches in the disjoint switch cover two subtypes `String | Int` and `Int | Bool`. When considered together those subtypes cover all possibilities for the value `x` (i.e. `x` can be either an integer, a string or a boolean, and the two cases cover all those possibilities). The *exhaustiveness* of the downcasts is what ensures that the downcasts are type-safe (that is they cannot fail at runtime). However, we also need to ensure that the two cases do not overlap to prevent ambiguity. In essence, in this simple setting of λ_u, checking that two types do not overlap amounts to check that there are no basic types (like `Int` or `Bool`) in common. In other words the only common subtypes should be bottom-like types.

Bottom-Like Types. *Bottom-like* types are types that are equivalent (i.e. both supertypes and subtypes) to $\bot$. In λ_u, there are infinitely many such types, and they all are uninhabited by values. According to the inductive definition shown at the top of Figure 3, they include the bottom type itself (via rule BL-BOT) and unions of two bottom-like types (via rule BL-OR), e.g. $\bot \vee \bot$. The correctness of our definition for bottom-like types is established by the following property:

▶ **Lemma 3** (Bottom-Like Soundness and Completeness). $\rfloor A \lfloor$ *if and only if* $\forall B,\ A <: B$.

Declarative Disjointness. The declarative definition for disjointness is as follows:

▶ **Definition 4** ($\bot$-Disjointness). $A * B ::= \forall\ C$, *if* $C <: A$ *and* $C <: B$ *then* $\rfloor C \lfloor$

That is, two types are disjoint if all their common subtypes are *bottom-like*. We give a few examples next, employing a bold font to highlight the types being compared for disjointness:

1. $\boldsymbol{A} =$ **Int**, $\boldsymbol{B} =$ **Int → Bool :** Int and Int → Bool are disjoint types. All common subtypes of Int and Int → Bool are *bottom-like* types, including $\bot$ and unions of $\bot$ types.
2. $\boldsymbol{A} =$ **Int ∨ Bool**, $\boldsymbol{B} =$ **⊥ :** Int ∨ Bool and $\bot$ are disjoint types. All common subtypes are *bottom-like*. In general, the type $\bot$ (or any *bottom-like* type) is disjoint to another type.
3. $\boldsymbol{A} =$ **Int**, $\boldsymbol{B} =$ **⊤ :** Int and $\top$ are not disjoint types because they share a common subtype Int which is not *bottom-like*. In general no type is disjoint to $\top$, except for *bottom-like* types. Also, one type is not disjoint with itself, unless it is *bottom-like*.
4. $\boldsymbol{A} =$ **Int → Bool**, $\boldsymbol{B} =$ **String → Char :** The types Int → Bool and String → Char are not disjoint, since we can find non-bottom-like types that are subtypes of both types. For instance $\top \rightarrow \bot$ is a subtype of both types. More generally, any two function types can never be disjoint: it is always possible to find a common subtype, which is not *bottom-like*.

Disjointness for Intersection Types. In essence, disjointness for λ_u is dual to the disjointness notion in λ_i [45], a calculus with disjoint intersection types. In λ_u, two types are disjoint if they do not share any common *subtype* which is not *bottom-like*. While in λ_i, two types are

$\boxed{\rfloor A \lfloor}$ *(Bottom-Like Types)*

$$\frac{}{\rfloor \bot \lfloor}\ \textsc{bl-bot} \qquad \frac{\rfloor A \lfloor \qquad \rfloor B \lfloor}{\rfloor A \vee B \lfloor}\ \textsc{bl-or}$$

$\boxed{A *_a B}$ *(Algorithmic Disjointness)*

$$\frac{}{A *_a \bot}\ \textsc{ad-btmr} \qquad \frac{}{\bot *_a A}\ \textsc{ad-btml} \qquad \frac{}{\mathsf{Int} *_a A \to B}\ \textsc{ad-intl} \qquad \frac{}{A \to B *_a \mathsf{Int}}\ \textsc{ad-intr}$$

$$\frac{}{\mathsf{Null} *_a \mathsf{Int}}\ \textsc{ad-null-intl} \qquad \frac{}{\mathsf{Int} *_a \mathsf{Null}}\ \textsc{ad-null-intr} \qquad \frac{}{\mathsf{Null} *_a A \to B}\ \textsc{ad-null-funl}$$

$$\frac{}{A \to B *_a \mathsf{Null}}\ \textsc{ad-null-funr} \qquad \frac{A *_a C \qquad B *_a C}{A \vee B *_a C}\ \textsc{ad-orl} \qquad \frac{A *_a B \qquad A *_a C}{A *_a B \vee C}\ \textsc{ad-orr}$$

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}\ \textsc{typ-int} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \mathsf{Null}}\ \textsc{typ-null} \qquad \frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \textsc{typ-sub}$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}\ \textsc{typ-app} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\ \textsc{typ-abs}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{typ-var}$$

$$\frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C \qquad A * B}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}\ \textsc{typ-switch}$$

Figure 3 Bottom-like types, algorithmic disjointness and typing for λ_u.

disjoint if they do not share any common *supertype* which is not *top-like* (i.e. equivalent to $\top$). While a disjoint switch provides deterministic behavior for downcasting, disjointness in intersection types prevents ambiguity in upcasting. In a type-safe setting, if two values v_1 and v_2 (of type A_1 and A_2) can both be upcasted to type B, then B must be a common supertype of A_1 and A_2. The disjointness restriction of A_1 and A_2 means they cannot have any *non-top-like* common supertype, so when the two values together upcasted to a type like Int, only one of them can contribute to the result. Prior work on disjoint intersection types is also helpful to find an algorithmic formulation of disjointness. Declarative disjointness does not directly lead to an algorithm. However, we can find an algorithmic formulation that employs dual rules to those for disjoint intersection types.

Algorithmic Disjointness. We present an algorithmic version of disjointness in the middle of Figure 3. Rules AD-BTMR and AD-BTML state that the $\bot$ type is disjoint to all types. Rules AD-INTL and AD-INTR state that Int and $A \to B$ are disjoint types. Algorithmic disjointness can further be scaled to more primitive disjoint types such as Bool and String by adding more rules similar to rules AD-INTL and AD-INTR for additional primitive types. Rules AD-NULL-INTL and AD-NULL-INTR state that Null and Int are disjoint types. Similarly,

rules AD-NULL-FUNL and AD-NULL-FUNR state that Null and $A \to B$ are disjoint types. Rules AD-ORL and AD-ORR are two symmetric rules for union types. Any type C is disjoint to an union type $A \vee B$ if C is disjoint to both A and B. We show that algorithmic disjointness is sound and complete with respect to its declarative specification (Definition 4).

▶ **Theorem 5** (Soundness and Completeness of Algorithmic Disjointness). *$A *_a B$ if and only if $A * B$.*

A natural property of λ_u is that if type A and type B are two disjoint types, then subtypes of A are disjoint to subtypes of B. This property dualises the *covariance of disjointness* property in calculi with disjoint intersection types [4].

▶ **Lemma 6** (Disjointness contravariance). *If $A * B$ and $C <: A$ and $D <: B$ then $C * D$.*

3.4 Typing

The typing rules are shown at the bottom of Figure 3. They are mostly standard. An integer has type Int, null has type Null and variable x gets type from the context. Rule TYP-APP is the standard rule for function application. Similarly, rule TYP-SUB and rule TYP-ABS are standard subsumption and abstraction rules respectively. The most interesting and novel rule is for *switch* expressions (rule TYP-SWITCH). It has four conditions. First, $\Gamma \vdash e : A \vee B$ ensures *exhaustiveness* of the cases in the switch: e must check against the types in the branches of the switch. The next two conditions ensure that branches of case expressions are well-typed and have type C, where the input variable is bound to type A and to type B respectively in the two branches. Finally, $A * B$ guarantees the *disjointness* of A and B. This forbids overlapping types for the branches of case expressions to avoid non-deterministic results. Since all the branches have type C, the whole switch expression has type C. Note that the two branches can have different return types. For example, if e_1 and e_2 have type Int and String respectively, the whole expression can have type Int $\vee$ String.

3.5 Operational Semantics

Now we discuss the small-step operational semantics of λ_u. An important aspect of this semantics is that union elimination is *type-directed*: types are used to pick the branch of the switch expression.

Figure 4 shows the operational semantics of λ_u. Rules STEP-APPL, STEP-APPR, and STEP-BETA are the standard call-by-value reduction rules for applications. Of particular interest are rules STEP-SWITCH, STEP-SWITCHL, and STEP-SWITCHR, which reduce the *switch* expressions. First, rule STEP-SWITCH reduces the case expression e, until it becomes a value v, at which point we must choose between the two branches of *switch*. We do so by inspecting the type of v: if the *approximate type* of v is a subtype of type of the left branch, then rule STEP-SWITCHL evaluates the left branch of the *switch* expression, or otherwise if it is a subtype of the type of the right branch, rule STEP-SWITCHR evaluates the right branch.

Note that the approximate type definition gives only a subtype of the actual type for a lambda value. This works, because the approximate type is only employed to allow the selection of a case with a function type, and in λ_u two function types can never be disjoint. Therefore, if there is a branch with a function type, then that must be the branch that applies to a lambda value. Note also that the program has been type-checked before hand, so we know that the static type of the value is compatible with the types on the branches. The subtyping condition in rules STEP-SWITCHL and STEP-SWITCHR is important, as it provides flexibility for the value to have various subtypes of A and B, instead of strictly having those

$\boxed{e \longrightarrow e'}$ *(Operational Semantics)*

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \textsc{step-appl} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}\ \textsc{step-appr} \qquad \frac{}{(\lambda x.e)\ v \longrightarrow e[x \rightsquigarrow v]}\ \textsc{step-beta}$$

$$\frac{e \longrightarrow e'}{\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}\ \textsc{step-switch}$$

$$\frac{\lfloor v \rfloor <: A}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]}\ \textsc{step-switchl}$$

$$\frac{\lfloor v \rfloor <: B}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]}\ \textsc{step-switchr}$$

Approximate Type $\lfloor v \rfloor$

$\lfloor i \rfloor$	$=$	Int
$\lfloor \lambda x.e \rfloor$	$=$	$\top \to \bot$
$\lfloor \mathsf{null} \rfloor$	$=$	Null

Figure 4 Operational semantics and approximate type definitions for λ_u.

types. Recall that the typing rule for *switch* (rule TYP-SWITCH) requires that types of left and right branches of a *switch* expression to be disjoint. This ensures that rules STEP-SWITCHL and STEP-SWITCHR cannot overlap, which, as we will see, is important for the operational semantics to be *deterministic*.

3.6 Type Soundness and Determinism

In this section, we prove that λ_u is type sound and deterministic. Type soundness is established by the type preservation and progress theorems. Type preservation (Theorem 7) states that types are preserved during reduction. Progress (Theorem 8) states that well typed programs never get stuck: a well typed expression e is either a value or it can reduce to some other expression e'.

▶ **Theorem 7** (Type Preservation). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : A$.

▶ **Theorem 8** (Progress). *If* $\cdot \vdash e : A$ *then either* e *is a value; or* $e \longrightarrow e'$ *for some* e'.

Determinism of λ_u (Theorem 10) ensures that a well-typed program reduces to a unique result. In particular, it guarantees that switch expressions are not order-sensitive: at any time, only one of the rules STEP-SWITCHL and STEP-SWITCHR can apply. The determinism of the switch expression relies on an essential property that a value cannot check against two disjoint types (Lemma 9).

▶ **Lemma 9** (Exclusivity of Disjoint Types). *If* $A * B$ *then* $\nexists\ v$ *such that both* $\Gamma \vdash v : A$ *and* $\Gamma \vdash v : B$ *holds.*

▶ **Theorem 10** (Determinism). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e_1$ *and* $e \longrightarrow e_2$ *then* $e_1 = e_2$.

3.7 An Alternative Specification for Disjointness

The current definition of disjointness (Definition 4) works well for the calculus presented in this section. But it is not the only possible formulation of disjointness. An equivalent formulation of disjointness is:

▶ **Definition 11** ($\wedge$-Disjointness). $A * B ::= \nexists\ C^{\circ},\ C^{\circ} <: A$ *and* $C^{\circ} <: B$

According to the new definition, two types are disjoint if they do not have common subtypes that are *ordinary*. Ordinary types (denoted by C°) are essentially those types that are primitive, such as integers and functions (see Figure 5 for a formal definition).

For the calculus presented in this section, we prove that the new definition is equivalent to the previous definition of disjointness.

▶ **Lemma 12** (Disjointness Equivalence). *Definition 11 ($\wedge$-Disjointness) is sound and complete to Definition 4 ($\bot$-Disjointness) in λ_u defined in this section.*

Why do we introduce the new definition of disjointness? It turns out that the previous definition is not sufficient when the calculus is extended with intersection types. As we will see, the new definition will play an important role in such variant of the calculus.

4 λ_u with Intersections, Distributive Subtyping and Nominal Types

In this section we extend λ_u with intersection types, nominal types and an enriched distributive subtyping relation. The study of an extension of λ_u with intersection types is motivated by the fact that most languages with union types also support intersection types (for example Ceylon, Scala or TypeScript). Furthermore, languages like Ceylon or Scala also support some form of distributive subtyping, as well as nominal types. Therefore it is important to understand whether those extensions can be easily added or whether there are some challenges. As it turns out, adding intersection types does pose a challenge, since the notion of disjointness inspired from disjoint intersection types [45] no longer works. Moreover subtyping relations with distributive subtyping add significant complexity, and we need an extension that supports nominal types as well. We show that desirable properties, including type soundness and determinism, are preserved in the extended version of λ_u. Moreover we prove that both disjointness and subtyping have sound, complete and decidable algorithms.

4.1 Syntax, Well-formedness and Ordinary Types

The syntax for this section mostly follows from Section 3, with the additional syntax given in Figure 5. The most significant difference and novelty in this section is the addition of intersection types $A \wedge B$ and an infinite set of nominal types. We use metavariable P to stand for nominal types. Expressions are extended with a new expression ($\mathsf{new}\ \mathsf{P}$) to create instances of nominal types. The expression $\mathsf{new}\ \mathsf{P}$ is also a value. Context Γ stays the same as in Section 3. We add a new context Δ, to track nominal types and their supertypes. For example, adding $\mathsf{P}_1 \leq \mathsf{P}_2$ to Δ declares a new nominal type P_1 that is a subtype of P_2. For a well-formed context, the supertype P_2 has to be declared before P_1. We also allow to declare a new nominal type P_1 with $\top$ as its supertype by adding $\mathsf{P}_1 \leq \top$ to Δ. Metavariable A°, B° and C° ranges over ordinary types [28]. There are four kinds of ordinary types: integers, null, function types and nominal types. Well-formed types and well-formedness of ordinary contexts Δ are shown in Figure 5.

Remark on Nominal Types. Note that our formulation of nominal types is simplified in two ways compared to languages like Java. Firstly, we do not consider arguments when building new expressions (i.e. we do not allow expressions like `new Person("John")`). Secondly, we also do not introduce class declarations, which would allow nominal types to be associated with method implementations. We follow a design choice for nominal types similar to Featherweight Java [38]. Featherweight Java uses a fixed size context for nominal types. Diamond inheritance is also not supported in Featherweight Java, and we follow that design

A, B, C	$::=$	$\ldots \mid A \wedge B \mid \mathsf{P}$
$A^\circ, B^\circ, C^\circ$	$::=$	$\mathsf{Int} \mid \mathsf{Null} \mid A \to B \mid \mathsf{P}$
e	$::=$	$\ldots \mid \mathsf{new}\ \mathsf{P}$
v	$::=$	$\ldots \mid \mathsf{new}\ \mathsf{P}$
Γ	$::=$	$\cdot \mid \Gamma, x : A$
Δ	$::=$	$\cdot \mid \Delta, \mathsf{P}_1 \leq \mathsf{P}_2 \mid \Delta, \mathsf{P} \leq \top$

$\boxed{\Delta \vdash A}$ *(Well-formed Types)*

$$\frac{}{\Delta \vdash \top}\ \textsc{wft-top} \qquad \frac{}{\Delta \vdash \bot}\ \textsc{wft-bot} \qquad \frac{}{\Delta \vdash \mathsf{Int}}\ \textsc{wft-int} \qquad \frac{}{\Delta \vdash \mathsf{Null}}\ \textsc{wft-null}$$

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}\ \textsc{wft-arrow} \qquad \frac{\mathsf{P} \in \mathit{dom}\ \Delta}{\Delta \vdash \mathsf{P}}\ \textsc{wft-prim} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \vee B}\ \textsc{wft-or}$$

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \wedge B}\ \textsc{wft-and}$$

$\boxed{\mathit{ok}\ \Delta}$ *(Well-formed Nominal Contexts)*

$$\frac{}{\mathit{ok}\ \cdot}\ \textsc{okp-empty} \qquad \frac{\mathit{ok}\ \Delta \qquad \mathsf{P} \notin \mathit{dom}\ \Delta}{\mathit{ok}\ \Delta, \mathsf{P} \leq \top}\ \textsc{okp-cons}$$

$$\frac{\mathit{ok}\ \Delta \qquad \Delta \vdash \mathsf{P}_2 \qquad \mathsf{P}_1 \notin \mathit{dom}\ \Delta}{\mathit{ok}\ \Delta, \mathsf{P}_1 \leq \mathsf{P}_2}\ \textsc{okp-sub}$$

Figure 5 Additional syntax and well-formedness.

choice as well. However, we believe that supporting diamond inheritance in our calculus is relatively easy. These simplifications keep the calculus simple, while capturing the essential features that matter for disjointness and the formalization of disjoint switches. Allowing for a more complete formulation of nominal types can be done in mostly standard ways.

4.2 Distributive Subtyping

Another interesting feature of this section is the addition of distributive subtyping to λ_u. Ceylon employs an enriched distributive subtyping relation [42] that is based on the B+ logic [50, 53]. To obtain an equivalent algorithmic formulation of subtyping, we employ the idea of *splittable types* [36], but extend that algorithm with the `Null` type and nominal types.

Distributive subtyping relation. Figure 6 shows a declarative version of distributive subtyping for λ_u with intersection and nominal types. Subtyping includes axioms for reflexivity (rule DS-REFL) and transitivity (rule DS-TRANS). Rules DS-TOP, DS-BOT, DS-ARROW, and DS-ORA have been discussed in Section 3. Rule DS-PRIM states that a nominal type is a subtype of type A if it is declared as subtype of A in Δ. Note that A can either be a nominal type or $\top$ under a well-formed context Δ. With the help of rule DS-TRANS, the subtyping of primitive types can also be constructed indirectly, e.g. $\mathsf{P}_1 \leq \top, \mathsf{P}_2 \leq \mathsf{P}_1, \mathsf{P}_3 \leq \mathsf{P}_2 \vdash \mathsf{P}_3 \leq \mathsf{P}_1$. Compared with the algorithmic formulation, having an explicit transitivity rule considerably

$\boxed{\Delta \vdash A \leq B}$ *(Declarative Subtyping with Distributivity)*

$$\frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A \leq A}\ \textsc{ds-refl} \qquad \frac{\Delta \vdash A \leq B \qquad \Delta \vdash B \leq C}{\Delta \vdash A \leq C}\ \textsc{ds-trans}$$

$$\frac{\Delta \vdash B_1 \leq A_1 \qquad \Delta \vdash A_2 \leq B_2}{\Delta \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \textsc{ds-arrow} \qquad \frac{ok\ \Delta \qquad \mathsf{P} \leq A \in \Delta}{\Delta \vdash \mathsf{P} \leq A}\ \textsc{ds-prim}$$

$$\frac{\Delta \vdash A_1 \leq B \qquad \Delta \vdash A_2 \leq B}{\Delta \vdash A_1 \vee A_2 \leq B}\ \textsc{ds-orA} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \leq A_1 \vee A_2}\ \textsc{ds-orB}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_2 \leq A_1 \vee A_2}\ \textsc{ds-orC} \qquad \frac{\Delta \vdash B \leq A_1 \qquad \Delta \vdash B \leq A_2}{\Delta \vdash B \leq A_1 \wedge A_2}\ \textsc{ds-andA}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \wedge A_2 \leq A_1}\ \textsc{ds-andB} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \wedge A_2 \leq A_2}\ \textsc{ds-andC}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2 \qquad \Delta \vdash B}{\Delta \vdash (A_1 \to B) \wedge (A_2 \to B) \leq (A_1 \vee A_2) \to B}\ \textsc{ds-distArrU} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A \leq \top}\ \textsc{ds-top}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2 \qquad \Delta \vdash B}{\Delta \vdash (A_1 \vee B) \wedge (A_2 \vee B) \leq (A_1 \wedge A_2) \vee B}\ \textsc{ds-distOr} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash \bot \leq A}\ \textsc{ds-bot}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A \qquad \Delta \vdash B_1 \qquad \Delta \vdash B_2}{\Delta \vdash (A \to B_1) \wedge (A \to B_2) \leq A \to (B_1 \wedge B_2)}\ \textsc{ds-distArr}$$

Figure 6 Distributive subtyping for λ_u with intersection types and nominal types.

simplifies the rules for nominal types. Rules DS-ORB and DS-ORC state that a subpart of a union type is a subtype of whole union type. Rule DS-ANDA states that a type A is a subtype of the intersection of two types B and C only if A is a subtype of both B and C. Rules DS-ANDB and DS-ANDC state that intersection type $A_1 \wedge A_2$ is a subtype of both A_1 and A_2 separately. Rule DS-DISTARR distributes function types over intersection types. It states that $(A \to B_1) \wedge (A \to B_2)$ is a subtype of $A \to (B_1 \wedge B_2)$. Rule DS-DISTARRU states that $(A_1 \to B) \wedge (A_2 \to B)$ is a subtype of $(A_1 \vee A_2) \to B$ type. Rule DS-DISTOR distributes intersections over unions.

Algorithmic Subtyping. Distributive rules make it hard to eliminate the transitivity rule. Our algorithmic formulation of distributive subtyping is based on a formulation using splittable types by Huang and Oliveira [36]. The basic idea is to view the distributive rules as some expansion of intersection and union types. For example, rule DS-DISTARR makes $A \to B_1 \wedge B_2$ and $(A \to B_1) \wedge (A \to B_2)$ mutual subtypes. Thus $A \to B_1 \wedge B_2$ is treated like $(A \to B_1) \wedge (A \to B_2)$ in the three intersection-related rules AS-ANDA, AS-ANDB, and AS-ANDC. Here we use $A \approxeq B \wedge C$ to denote that type A can be split into B and C (and therefore, A is equivalent to $B \wedge C$) according to the procedure designed by Huang and Oliveira. Union and union-like types (e.g. $(A_1 \vee A_2) \wedge B \approxeq A_1 \wedge B \vee A_2 \wedge B$) are handled in similar way in rules AS-ORA, AS-ORB, and AS-ORC. For further details of algorithmic subtyping we refer to their paper.

$\boxed{\Delta \vdash A <: B}$ *(Algorithmic Subtyping with Distributivity)*

$$\frac{\Delta \vdash B_1 <: A_1 \qquad \Delta \vdash A_2 <: B_2}{\Delta \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{as-arrow} \qquad \frac{ok\ (\Delta, \mathsf{P}_1 <: \mathsf{P}_2) \qquad \Delta \vdash \mathsf{P}_2 <: \mathsf{P}_3}{\Delta, \mathsf{P}_1 <: \mathsf{P}_2 \vdash \mathsf{P}_1 <: \mathsf{P}_3}\ \textsc{as-primEq}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: A}\ \textsc{as-refl} \qquad \frac{ok\ (\Delta, \mathsf{P}_2 <: A) \qquad \mathsf{P}_1 \neq \mathsf{P}_2 \qquad \Delta \vdash \mathsf{P}_1 <: \mathsf{P}_3}{\Delta, \mathsf{P}_2 <: A \vdash \mathsf{P}_1 <: \mathsf{P}_3}\ \textsc{as-primNeq}$$

$$\frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash A_1 <: B \qquad \Delta \vdash A_2 <: B}{\Delta \vdash A <: B}\ \textsc{as-orA} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: \top}\ \textsc{as-top}$$

$$\frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash B <: A_1}{\Delta \vdash B <: A}\ \textsc{as-orB} \qquad \frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash B <: A_2}{\Delta \vdash B <: A}\ \textsc{as-orC}$$

$$\frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash B <: A_1 \qquad \Delta \vdash B <: A_2}{\Delta \vdash B <: A}\ \textsc{as-andA} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash \bot <: A}\ \textsc{as-bot}$$

$$\frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash A_1 <: B}{\Delta \vdash A <: B}\ \textsc{as-andB} \qquad \frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash A_2 <: B}{\Delta \vdash A <: B}\ \textsc{as-andC}$$

Figure 7 Algorithmic subtyping for λ_u with distributivity, intersection and nominal types.

Subtyping Nominal Types. However, Huang and Oliveira's algorithm does not account for `Null` and nominal types. We add the nominal context Δ in the subtyping judgment and extend the subtyping algorithm with `Null` and nominal types. Nominal types are not splittable, and their subtyping relation is defined by the transitive closure of the context. They are supertypes of $\bot$ and subtypes of $\top$, but not related with other primitive types like Int and Null. So for nominal types, we mainly focus on checking the subtyping relationship among them in our algorithm. Given a well-formed context, any nominal type P appears only once in a subtype position as an explicit declaration for P, and its direct supertype, if is not $\top$, must be declared before P. Thus if $\Delta \vdash \mathsf{P}_1 <: \mathsf{P}_2$ holds, either P_2 is introduced before P_1 in Δ, or they are the same type, in which case the goal can be solved by rule AS-REFL. For the other cases, we recursively search for P_1 in all subtype positions of the context Δ (rule AS-PRIMNEQ). When we find P_1, we check its direct supertype. If it is $\top$, no other nominal types can be supertypes of P_1. So in rule AS-PRIMEQ, we only consider when the direct supertype is another primitive P_2. For P_3 to be a supertype of P_1, it must either equal to P_2, or it is related to P_2 by the smaller context. In either case, we can prove that P_3 is a supertype of the direct supertype of P_1.

Inversion Lemmas for Type Soundness. Having an algorithmic formulation of subtyping is useful to prove several inversion lemmas that are used in the type soundness proof. For instance, it allows us to prove the following lemma:

▶ **Lemma 13** (Inversion on Function Types). *If* $\Delta \vdash A_1 \to A_2 <: B_1 \to B_2$ *then* $\Delta \vdash B_1 <: A_1$ *and* $\Delta \vdash A_2 <: B_2$.

While the additional distributive rules make function types more flexible, they retain the contravariance of argument types and covariance of return types. In addition, we show the formulation is sound and complete to the declarative subtyping and it is decidable whether a subtyping judgment holds under a given context.

▶ **Lemma 14** (Equivalence of subtyping). *$\Delta \vdash A \leq B$ if and only if $\Delta \vdash A <: B$.*

▶ **Lemma 15** (Decidability of subtyping). *$\Delta \vdash A \leq B$ is decidable.*

4.3 Disjointness Specification

Disjointness is another interesting aspect of the extension of λ_u. Unfortunately, Definition 4 does not work with intersection types. In what follows, we first explain why Definition 4 does not work, and then we show how to define disjointness in the presence of intersection types.

Bottom-like types, intersection types and disjointness. Recall that disjointness in Section 3 (Definition 4) depends on bottom-like types, where two types are disjoint only if all their common subtypes are bottom-like. However, this definition no longer works with the addition of intersection types. According to the subtyping rule for intersection types, any two types have their intersection as one common subtype. For non-bottom-like types, their intersection is also not bottom-like. For example, type Int and type Bool now have a non-bottom like subtype $\mathsf{Int} \wedge \mathsf{Bool}$. In other words, the disjointness definition fails, since we can always find a common non-bottom-like subtype for any two (non-bottom-like) types.

A possible solution: the Ceylon approach. A possible solution for this issue is to add a subtyping rule which makes intersections of disjoint types subtypes of $\bot$.

$$\frac{A * B}{A \wedge B <: \bot}\ \textsc{s-disj}$$

This rule is adopted by the Ceylon language [42]. With the rule S-DISJ now the type $\mathsf{Int} \wedge \mathsf{Bool}$ would be a bottom-like type, and the definition of disjointness used in Section 3 could still work. The logic behind this rule is that if we interpret types as sets of values, and intersection as set intersection, then intersecting disjoint sets is the empty set. In other words, we would get a type that has no inhabitants. For instance the set of all integers is disjoint to the set of all booleans, and the intersection of those sets is empty. However we do not adopt the Ceylon solution here for two reasons:

1. Rule S-DISJ complicates the system because it adds a mutual dependency between subtyping and disjointness: disjointness is defined in terms of subtyping, and subtyping now uses disjointness as well in rule S-DISJ. This creates important challenges for the metatheory. In particular, the completeness proof for disjointness becomes quite challenging.
2. Additionally, the assumption that intersections of disjoint types are equivalent to $\bot$ is too strong for some calculi with intersection types. If a merge operator [49] is allowed in the calculus, intersection types can be inhabited with values (for example, in λ_i [45], the type $\mathsf{Int} \wedge \mathsf{Bool}$ is inhabited by $1 ,, \mathsf{True}$). Considering those types bottom-like would lead to a problematic definition of subtyping, since some bottom-like types (those based on disjoint types) would be inhabited.

For those reasons we adopt a different approach in λ_u. Nevertheless, in the extended version of the paper we show that it is possible to create an extension of λ_u that includes (and in fact generalizes) the Ceylon-style rule S-DISJ.

Lowest Ordinary Subtypes (LOS) $|A|_\Delta$

$\lvert\top\rvert_\Delta$	$=$	$\{\mathsf{Int}, \top \to \bot, \mathsf{Null}\} \cup \mathrm{dom}\ \Delta$
$\lvert\bot\rvert_\Delta$	$=$	$\{\}$
$\lvert\mathsf{Int}\rvert_\Delta$	$=$	$\{\mathsf{Int}\}$
$\lvert A \to B\rvert_\Delta$	$=$	$\{\top \to \bot\}$
$\lvert A \vee B\rvert_\Delta$	$=$	$\lvert A\rvert_\Delta \cup \lvert B\rvert_\Delta$
$\lvert A \wedge B\rvert_\Delta$	$=$	$\lvert A\rvert_\Delta \cap \lvert B\rvert_\Delta$
$\lvert\mathsf{Null}\rvert_\Delta$	$=$	$\{\mathsf{Null}\}$
$\lvert\mathsf{P}\rvert_\Delta$	$=$	$\{\mathsf{P}\} \cup \Delta(\mathsf{P})$

Nominal Subtypes $\boxed{\Delta(A)}$

$$\cdot(A) = \{\}$$

$$(\Delta', \mathsf{P} \leq B)(A) = \begin{cases} \{\mathsf{P}\} \cup \Delta'(A) \textbf{ if } \mathsf{P} \leq A \in \Delta \\ \Delta'(A) \textbf{ otherwise} \end{cases}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{new}\ \mathsf{P} : \mathsf{P}}\ \text{PTYP-PRIM}$$

Figure 8 Lowest ordinary subtypes function and additional typing rule for λ_u with intersection types and nominal types.

Disjointness based on ordinary types to the rescue. To solve the problem with the disjointness specification, we resort to the alternative definition of disjointness presented in Section 3.7. Note that now the disjointness definition also contains Δ as an argument to account for nominal types.

▶ **Definition 16** ($\wedge$-Disjointness). $\Delta \vdash A * B ::= \nexists\ C^\circ,\ \Delta \vdash C^\circ <: A$ *and* $\Delta \vdash C^\circ <: B$.

Interestingly, while in Section 3 such definition was equivalent to the definition using bottom-like types, this is no longer the case for λ_u with intersection types. To see why, consider again the types Int and Bool. Int and Bool do not share any common ordinary subtype. Therefore, Int and Bool are disjoint types according to Definition 16. We further illustrate Definition 16 with a few concrete examples:

1. $\boldsymbol{A = \mathsf{Int} \vee \mathsf{Bool},\ B = \bot}$ **:** Since there is no ordinary type that is a subtype of both $\mathsf{Int} \vee \mathsf{Bool}$ and $\bot$, $\mathsf{Int} \vee \mathsf{Bool}$ and $\bot$ are disjoint types. In general, the $\bot$ type is disjoint to all types because $\bot$ does not have any ordinary subtype.
2. $\boldsymbol{A = \mathsf{Int} \wedge \mathsf{Bool},\ B = \mathsf{Int} \vee \mathsf{Bool}}$ **:** There is no ordinary type that is a subtype of both $\mathsf{Int} \wedge \mathsf{Bool}$ and $\mathsf{Int} \vee \mathsf{Bool}$. Therefore, $\mathsf{Int} \wedge \mathsf{Bool}$ and $\mathsf{Int} \vee \mathsf{Bool}$ are disjoint types. In general, an intersection of two disjoint types ($\mathsf{Int} \wedge \mathsf{Bool}$ in this case) is always disjoint to all types.

4.4 Algorithmic Disjointness

The change in the disjointness specification has a significant impact on an algorithmic formulation. In particular, it is not obvious at all how to adapt the algorithmic formulation in Figure 3. To obtain a sound, complete and decidable formulation of disjointness, we employ the novel notion of *lowest ordinary subtypes.*

Lowest ordinary subtypes ($|A|_\Delta$). Figure 8 shows the definition of *lowest ordinary subtypes* (LOS) ($|A|_\Delta$). LOS is defined as a function which returns a set of ordinary subtypes of the given input type. No ordinary type is a subtype of $\bot$. The only ordinary subtype of Int is Int itself. The function case is interesting. Since no two functions are disjoint in the calculus proposed in this paper, the case for function types $A \to B$ returns $\top \to \bot$. This type is the least ordinary function type, which is a subtype of all function types. Lowest ordinary subtypes of $\top$ are Int, $\top \to \bot$, Null and all the nominal types defined in Δ. In the case of union types $A \vee B$, the algorithm collects the LOS of A and B and returns the union of the two sets. For intersection types $A \wedge B$ the algorithm collects the LOS of A and B and returns the intersection of the two sets. The lowest ordinary subtype of Null is Null itself. Finally, the LOS of P is the union of P itself with all subtypes of P defined in Δ. Note that LOS is defined as a structurally recursive function and therefore its decidability is immediate.

Algorithmic disjointness. With LOS, an algorithmic formulation of disjointness is straightforward:

▶ **Definition 17.** $\Delta \vdash A *_a B ::= |A|_\Delta \cap |B|_\Delta = \{\}$.

The algorithmic formulation of disjointness in Definition 17 states that two types A and B are disjoint under the context Δ if they do not have any common lowest ordinary subtypes. In other words, the set intersection of $|A|_\Delta$ and $|B|_\Delta$ is the empty set. Note that this algorithm is naturally very close to Definition 16.

Soundness and completeness of algorithmic disjointness. Next, we show that disjointness algorithm is sound and complete with respect to disjointness specifications (Theorem 18). Soundness and completeness of LOS are essential to prove Theorem 18. Both of these properties are shown in Lemma 19 and Lemma 20 respectively.

▶ **Theorem 18** (Disjointness Equivalence). *$\Delta \vdash A *_a B$ if and only if $\Delta \vdash A * B$.*

▶ **Lemma 19** (Soundness of $|A|_\Delta$). *$\forall$ well-formed Δ and A and B that are well-formed under Δ, if $B \in |A|_\Delta$, then $\Delta \vdash B <: A$.*

▶ **Lemma 20** (Completeness of $|A|_\Delta$). *$\forall$ A B°, if $\Delta \vdash B^\circ <: A$, then $B^\circ \in |A|_\Delta$, or B° is an arrow type and $\top \to \bot \in |A|_\Delta$.*

4.5 Typing, Semantics and Metatheory

Both typing and the operational semantics are parameterized by the nominal context Δ. The typing rules are extended with a rule for nominal types rule PTYP-PRIM as shown at the right side in Figure 8. The typing rule PTYP-PRIM states that under a well-formed context Δ and well-formed type P, $\mathsf{new}\ \mathsf{P}$ has type P. No additional reduction rule is required because $\mathsf{new}\ \mathsf{P}$ is a value. However, the rules STEP-SWITCHL and STEP-SWITCHR require Δ because they do a subtyping check. We illustrate the updated rule STEP-SWITCHL next:

$$\frac{\Delta \vdash \lfloor v \rfloor <: A}{\Delta \vdash \mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]}\ \text{NSTEP-SWITCHL}$$

Rule STEP-SWITCHR is updated similarly. All the other rules are essentially the same as in Section 3, modulo the extra nominal context Δ.

Example. Assuming a context $\Delta = \mathsf{Person} \leq \top, \mathsf{Student} \leq \mathsf{Person}, \mathsf{Robot} \leq \top, \mathsf{y} : \mathsf{Person} \mid \mathsf{Robot}$ and $\mathsf{x} : \mathsf{Student}$, we could write the following two switches:

```
switch(y)      // Accepted!
  (z : Person)  -> False
  (z : Robot)   -> True
```

```
switch (x)     // Rejected!
  (z : Person)  -> False
  (z : Student) -> True
```

In the above code, the first switch, using `y` is accepted, while the second one (using `x`) is rejected because the types overlap in that case.

Key Properties. We proved that λ_u with intersection types, nominal types and subtyping distributivity preserves type soundness and determinism.

▶ **Theorem 21** (Type Preservation). *If $\Delta; \Gamma \vdash e : A$ and $e \longrightarrow e'$ then $\Delta; \Gamma \vdash e' : A$.*

▶ **Theorem 22** (Progress). *If $\Delta; \cdot \vdash e : A$ then either e is a value; or e can take a step to e'.*

▶ **Theorem 23** (Determinism). *If $\Delta; \Gamma \vdash e : A$ and $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

5 Related Work

Union types. Union types were first introduced by MacQueen et al. [41]. They proposed a typing rule that eliminates unions implicitly. The rule breaks type preservation under the conventional reduction strategy of the lambda calculus. Barbanera et al. [9] solved the problem by reducing all copies of the same redex in parallel. Dunfield and Pfenning [30, 29] took another approach to support mutable references. They restricted the elimination typing rule to only allow a single occurrence of a subterm with a union type when typing an expression. Pierce [47] proposed a novel single-branch case construct for unions. As pointed by Dunfield and Pfenning, compared to the single occurrence approach, the only effect of Pierce's approach is to make elimination explicit.

Union types and elimination constructs based on types are widely used in the context of XML processing languages [35, 10], and have inspired proposals for object oriented languages [37]. Generally speaking, the elimination constructs in those languages offer a first-match semantics, where cases can overlap and reordering the cases may change the semantics of the program. This is in contrast to our approach. Union types have also been studied in the context of XDuce programming language [35]. XDuce employs regular expression types. Pattern matching can be on expressions and types in XDuce. Expressions are considered as special cases of types. CDuce [10] is an extension of XDuce. Work on the more foundational aspects of CDuce, and in particular on *semantic subtyping* [31] and set-theoretic types, also employs a form of first-match semantics elimination construct, though in a different form. In particular, work by Castagna et al. [18, 20] proposes a conditional construct that can test whether a value matches a type. If it matches then the first branch is executed and the type for the value is refined. Otherwise, the second branch is executed and the type of the value is refined to be the negation of the type (expressing that the value does not have such type). Union types are also studied in the context of semantic subtyping and object-oriented calculi [6, 5, 27] which focus on designing subtyping algorithms to employ semantic subtyping in OOP. In contrast, we study a deterministic and type-safe switch construct for union elimination.

Muehlboeck and Tate [42] give a general framework for subtyping with intersection and union types. They illustrate the significance of their framework using the Ceylon programming language. The main objective of their work is to define a generic framework for deriving subtyping algorithms for intersection and union types in the presence of various distributive subtyping rules. For instance, their framework could be useful to derive an algorithmic formulation for the subtyping relation presented in Figure 6. They also briefly cover disjointness in their work. As part of their framework, they can also check disjointness given some disjointness axioms. For instance, for λ_u, such axioms could be similar to rule AD-BTMR or rule AD-INTL in Figure 3. However, they do not have a formal specification of disjointness. Instead they assume that some sound specification exists and that the axioms respect such specification. If some unsound axioms are given to their framework (say $\mathsf{Int} *_a \mathsf{Int}$) this would lead to a problematic algorithm for checking disjointness. In our work we provide specifications for disjointness together with sound and complete algorithmic formulations. In addition, unlike us, they do not study the semantics of disjoint switch expressions.

Occurrence Typing. Occurrence typing or flow typing [51] specializes or refines the type of variable in a type test. An example of occurrence typing is:

```
Integer occurrence (Integer | String val) {
  if (val is Integer) { return val+1; }
  else                { return toInt(val)+2; }
}
```

In such code, `val` initially has type Int $\vee$ String. The conditional checks if the `val` is of type Int. If the condition succeeds, it is safe to assume that `val` is of type Int, and the type of `val` is refined in the branch to be Int. Otherwise, it is safe to assume that `val` is of type String, in the other branch (and the type is refined accordingly). The motivation to study occurrence typing was to introduce typing in dynamically typed languages. Occurrence typing was further studied by Tobin-Hochstadt and Felleisen [52], which resulted into the development of Typed Racket. Variants of occurrence typing are nowadays employed in mainstream languages such as TypeScript, Ceylon or Flow. Castagna et al. [21] extended occurrence typing to refine the type of generic expressions, not just variables. They also studied the combination with gradual typing. Occurrence typing in a conditional construct, such as the above, provides an alternative means to eliminate union types using a first-match semantics. That is the order of the type tests determines the priority.

Nullable Types. Nullable types are types which may have the `null` value. Recently, Nieto et al. [43] proposed an approach with explicit nulls in Scala using union types. The Ceylon language has implemented a similar approach for a few years now. However our's and Ceylon's approaches are based on disjoint switches to test for nullability, while Nieto et al.'s [43] approach is based on a simplified form of occurrence typing.

Various approaches have been proposed to deal with nullability such as `T?` in Kotlin [40], Swift [7] and Flow [25]. The Checker Framework [46] is another line of related work to detect null pointer deferences in Java programs. Banerjee et al. [8] proposes an approach to explicitly associate nullable and non-nullable properties with expressions in Java. However, differently from our work, in those approaches nullable types are not encoded with union types. Blanvillain et al. [14] study a notion of match types for type level programming. They also employ a notion of disjointness in match types and can encode nullable types. However, they provide match types at the type level and do not use them for union elimination. Furthermore, they do not study intersection and union types formally. In contrast, we provide a term level switch construct for union elimination.

Disjoint Intersection Types. Disjoint intersection types were first studied by Oliveira et al. [45] in the λ_i calculus to give a coherent calculus for intersection types with a merge operator. The notion of disjointness used in λ_u, discussed in Section 3, is inspired by the notion of disjointness of λ_i. In essence, disjointness in λ_u is the dual notion: while in λ_i two types are disjoint if they only have *top-like* supertypes, in λ_u two types are disjoint if they only have *bottom-like* subtypes. *Disjoint polymorphism* [4] has been studied for calculi with disjoint intersection types.

None of calculi with disjoint intersection types [45, 11, 4, 12] in the literature includes union types. One interesting discovery of our work is that the presence of both intersections and unions in a calculus can affect disjointness. In particular, as we have seen in Section 4, adding intersection types required us to change disjointness. The notion of disjointness that was derived from λ_i stops working in the presence of intersection types. Interestingly, a similar issue happens when union types are added to a calculus with disjoint intersection types. If disjointness of two types `A` and `B` is defined to be that such types can only have top-like types, then adding union types immediately breaks such definition. For example, the types Int and Bool are disjoint but, with union types, Int $\vee$ Bool is a common supertype that is not top-like. We conjecture that, to add union types to disjoint intersection types, we can use the following definition of disjointness:

▶ **Definition 24.** $A * B ::= \nexists\, C^\circ,\ A <: C^\circ$ *and* $B <: C^\circ$.

which is, in essence, the dual notion of the definition presented in Section 4. Under this definition Int and Bool would be disjoint since we cannot find a common ordinary supertype (and $\mathsf{Int} \vee \mathsf{Bool}$ is a supertype, but it is not ordinary). Furthermore, there should be a dual notion to LOS, capturing the greatest ordinary supertypes. Moreover, if a calculus includes both disjoint switches and a merge operator, then the two notions of disjointness must coexist in the calculus. This will be an interesting path of exploration for future work.

Overloading. Union and intersection types also provide a form of function overloading or ad-hoc polymorphism using the switch and type-based case analysis. A programmer may define the argument type to be a union type. By using type-based case analysis, it is possible to execute different code for each specific type of input. Intersection types have also been studied for function overloading. For example, a function with type $\mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}$ can take input values either of type Int or Bool. In such case, it returns either Int or Bool depending upon the input type. Function overloading [19, 17, 56] has been studied in detail in the literature. Wadler and Blott [56] studied type classes as an alternative way to provide overloading based on parametric polymorphism.

6 Conclusion and Future Work

This work develops the union calculus (λ_u) with union types and a type-based union elimination construct based on disjointness. We presented the operational semantics of the calculus, and showed type-soundness and determinism. Disjointness plays a crucial role for the determinism result, as it ensures that only one branch in the switch elimination construct can apply for any given value. A nice aspect of the work was that we were able to adapt the notion of disjointness used in disjoint intersection types to our variant of λ_u with union types. We believe that this reinforces fundamental connections between union and intersection types via duality. The addition of intersection types to λ_u lead to some interesting discoveries. In particular, it showed that the notion of disjointness that we were able to formulate, inspired by the work on disjoint intersection types, breaks. This is not showing that the duality stops working. Instead, it shows that the combination of intersections and unions in the same system affects disjointness. As discussed in Section 5, adding union types to calculi with disjoint intersection types leads to a similar problem, and the solution in λ_u can inspire solutions for adding union types to disjoint intersection types.

We plan to extend λ_u for practical programming languages with more advanced features. An interesting line of research for λ_u is to study the addition of the merge operator, which calculi with disjoint intersection types include. The main challenge is that types such as $\mathsf{Int} \wedge \mathsf{Bool}$ become inhabited. It could also be interesting to study a variant of λ_u that uses a best-match approach based on the dynamic type. This would relate to the extensive line of research on *multi-methods* [23] and *multiple dispatching* [26]. Finally, a current limitation of our approach is that it relies on a global context for nominal types. This enables some simplifications, since we can search the global nominal environment for subtypes. However this assumption breaks in a setting where new nominal types can be added. Ceylon solves this issue in a modular way using `of` clauses that enumerate all the possible subtypes that a class can have. It would be interesting to adopt this approach to enable the addition of new nominal types.

References

1 Disjointness in ceylon. URL: http://web.mit.edu/ceylon_v1.3.3/ceylon-1.3.3/doc/en/spec/html_single.

2 Overloading in ceylon. URL: https://github.com/ceylon/ceylon-spec/issues/73.

3 Union types in typescript. URL: https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html.

4 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.

5 Davide Ancona and Andrea Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. In *European Conference on Object-Oriented Programming*, pages 282–307. Springer, 2014.

6 Davide Ancona and Andrea Corradi. Semantic subtyping for imperative object-oriented languages. *ACM SIGPLAN Notices*, 51(10):568–587, 2016.

7 Inc Apple. Swift language guide, 2021. URL: https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html.

8 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 740–750, 2019.

9 Franco Barbanera, Mariangiola Dezaniciancaglini, and Ugo Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

10 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003.

11 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

12 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In Luís Caires, editor, *Programming Languages and Systems*, pages 381–409, Cham, 2019. Springer International Publishing.

13 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

14 Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498698.

15 R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. Technical Report CSR-62-80, Computer Science Dept, Univ. of Edinburgh, 1981.

16 Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, pages 273–280, 1989.

17 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

18 Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–199, 2005.

19 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

20 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

21 Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. Revisiting occurrence typing. *arXiv preprint arXiv:1907.05590*, 2019.

22 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: Part 1: Syntax, semantics, and evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 5–17, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2535838.2535840`.

23 Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615, pages 33–56. Springer-Verlag, 1992.

24 Avik Chaudhuri. Flow: a static type checker for javascript. *SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

25 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

26 Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 130–145, 2000.

27 Ornela Dardha, Daniele Gorla, and Daniele Varacca. Semantic subtyping for objects and classes. In *Formal Techniques for Distributed Systems*, pages 66–82. Springer, 2013.

28 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 198–208, 2000.

29 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.

30 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *International Conference on Foundations of Software Science and Computation Structures*, pages 250–266. Springer, 2003.

31 Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE, 2002.

32 Jacques Garrigue. Programming with polymorphic variants. In *ML workshop*, 1998.

33 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java language specification, 2021. URL: `https://docs.oracle.com/javase/specs/jls/se14/html/index.html`.

34 Eric Gunnerson. Nullable types. In *A Programmer's Guide to C# 5.0*, pages 247–250. Springer, 2012.

35 Haruo Hosoya and Benjamin C Pierce. Xduce: A statically typed xml processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

36 Xuejing Huang and Bruno C d S Oliveira. Distributing intersection and union types with splits and duality (functional pearl). *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–24, 2021.

37 Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1435–1441, 2006.

38 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. `doi:10.1145/503502.503505`.

39 Gavin King. The ceylon language specification, version 1.0, 2013.

40 Foundation Kotlin. Kotlin programming language, 2021. URL: `https://kotlinlang.org/`.

41 David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 165–174, 1984.

42 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

43 Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with Explicit Nulls. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 25:1–25:26, 2020.

44 Martin Odersky. Scala 3: A next generation compiler for scala, 2021. URL: `https://dotty.epfl.ch`.

45 Bruno C. d. S. Oliveira, Zhiyuan Shi, and Joao Alpuim. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 364–377, 2016.

46 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

47 Benjamin C Pierce. Programming with intersection types, union types. Technical report, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

48 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

49 John C Reynolds. Preliminary design of the programming language forsythe, 1988.

50 Richard Routley and Robert K Meyer. The semantics of entailment—iii. *Journal of philosophical logic*, 1(2):192–208, 1972.

51 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

52 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 117–128, 2010.

53 Steffen van Bakel, Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Yoko Motohoma. The minimal relevant logic and the call-by-value lambda calculus. Technical report, Citeseer, 2000.

54 Adriaan Van Wijngaarden, Barry J Mailloux, John EL Peck, Cornelius HA Koster, M Sintzoff, CH Lindsey, LGLT Meertens, and RG Fisker. Report on the algorithmic language algol 68. *Numerische Mathematik*, 14(1):79–218, 1969.

55 Adriaan van Wijngaarden, Barry James Mailloux, John Edward Lancelot Peck, Cornelis HA Koster, CH Lindsey, M Sintzoff, Lambert GLT Meertens, and RG Fisker. *Revised report on the algorithmic language Algol 68*. Springer Science & Business Media, 2012.

56 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.

57 Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

A Further Extensions and Discussion

The calculus introduced in Section 3 is a simple foundational lambda calculus with union types, similar to prior work on union types and their elimination forms [10, 29, 22]. In Section 4 we extend λ_u with various interesting features including intersection types, nominal types and subtyping distributivity, inspired by Ceylon, which has similar features. In this section we discuss two more practical extensions:

- **Disjoint Polymorphism:** The first extension is an extension with a form of *disjoint polymorphism* [4], which allows the specification of disjointness constraints for type variables. Although Ceylon supports polymorphism, it does not support disjoint polymorphism. The extension with disjoint polymorphism is inspired by the work on disjoint intersection types, where disjoint polymorphism has been proposed to account for disjointness in a polymorphic language.

$$\begin{array}{rcl} A, B, C & ::= & \dots \mid \alpha \mid \forall(\alpha * G).B \\ e & ::= & \dots \mid e\,A \mid \Lambda(\alpha * G).e \\ v & ::= & \dots \mid \Lambda(\alpha * G).e \\ \Gamma & ::= & \dots \mid \Gamma, \alpha * G \\ G & ::= & \top \mid \bot \mid \mathsf{Int} \mid \mathsf{Null} \mid A \to B \\ & & \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \forall(\alpha * G).B \end{array}$$

Lowest Ordinary Subtypes (LOS) $|A|_{\Delta;\Gamma}$

$$\begin{array}{lcl} \dots & = & \dots \\ |\forall(\alpha * G).B|_{\Delta;\Gamma} & = & \{\forall(\alpha * \bot).\bot\} \\ |\alpha|_{\Delta;\Gamma} & = & (|\top|_{\Delta;\Gamma}) \text{ - } (|G|_{\Delta;\Gamma}) \\ & & \text{where } \alpha * G \in \Gamma \end{array}$$

$\boxed{\Delta; \Gamma \vdash A <: B}$ *(Additional subtyping rules)*

$$\frac{ok\ \Delta \qquad \Delta; \Gamma \vdash \alpha}{\Delta; \Gamma \vdash \alpha <: \alpha}\ \text{POLYS-TVAR} \qquad \frac{\Delta; \Gamma \vdash G_1 <: G_2 \qquad \Delta; \Gamma, \alpha * G_2 \vdash B_1 <: B_2}{\Delta; \Gamma \vdash \forall(\alpha * G_1).B_1 <: \forall(\alpha * G_2).B_2}\ \text{POLYS-ALL}$$

$\boxed{\Delta; \Gamma \vdash e : A}$ *(Additional typing rules)*

$$\frac{\Delta; \Gamma \vdash e : \forall(\alpha * G).C \qquad \Delta; \Gamma \vdash G_1 * G}{\Delta; \Gamma \vdash e\,G_1 : C[\alpha \rightsquigarrow G_1]}\ \text{PTYP-TAP} \qquad \frac{\Delta; \Gamma, \alpha * G \vdash e : B}{\Delta; \Gamma \vdash \Lambda(\alpha * G).e : \forall(\alpha * G).B}\ \text{PTYP-TABS}$$

$\boxed{\Delta; \Gamma \vdash e \longrightarrow e'}$ *(Additional reduction rules)*

$$\frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash e\,B \longrightarrow e'\,B}\ \text{POLYSTEP-TAPPL} \qquad \frac{}{\Delta; \Gamma \vdash (\Lambda(\alpha * G).e)\,B \longrightarrow e[\alpha \rightsquigarrow B]}\ \text{POLYSTEP-TAPP}$$

Figure 9 Syntax, additional typing, subtyping, and reduction rules for λ_u with polymorphism.

- **A Special Subtyping Rule for Empty Types:** The second extension that we discuss is an alternative subtyping formulation with a special subtyping rule for empty types, which follows the Ceylon approach.

Note that both extensions above have also been formalized in Coq and proved type-sound and deterministic. In addition, we also have a brief discussion about implementation considerations.

A.1 Polymorphism

Polymorphism is an essential feature of almost all the modern programming languages. In this section we discuss an extension of λ_u with parametric polymorphism along with intersection and nominal types. The interesting aspect about this extension is the presence of disjointness constraints. For example, in λ_u with polymorphism a polymorphic disjoint switch such as: $\Gamma, \alpha * \mathsf{Int} \vdash \mathsf{switch}\ e\ \{(x : \mathsf{Int}) \to \mathsf{true}, (y : \alpha) \to \mathsf{false}\}$ is accepted. It is safe to use Int and α in alternative branches in a switch in this example. The disjointness constraint in the context $(\Gamma, \alpha * \mathsf{Int})$ on type variable α ensures that α must only be instantiated with types disjoint to Int. Thus an instantiation of α with Null or $A \to B$ is allowed. Whereas, an instantiation of α with Int is rejected by the type system.

Syntax. Figure 9 shows the extension in the syntax of λ_u with polymorphism. Types are extended with type variables α and disjoint quantifiers $\forall(\alpha * G).B$. $\forall(\alpha * G).B$ is also an ordinary type. The reader can think of this extension in the context of bounded quantification [17, 16] where bounded quantifiers $(\forall(\alpha <: A).B)$ are replaced by disjoint quantifiers $(\forall(\alpha * G).B)$. Bounded quantification imposes a subtyping restriction on type variables, whereas disjoint quantification imposes disjointness restriction on type variables.

Disjoint quantification only allows the instantiation of disjoint types. For example, $\forall(\alpha <: \mathsf{Int} \vee \mathsf{Bool}).\alpha$ allows α to be instantiated only with subtypes of $\mathsf{Int} \vee \mathsf{Bool}$ and restricts all other types. Whereas, $\forall(\alpha * \mathsf{Int} \vee \mathsf{Bool}).\alpha$ restricts all the instantiations of α which share an ordinary subtype with $\mathsf{Int} \vee \mathsf{Bool}$. In other words, the permitted instantiations of α are the types disjoint to $\mathsf{Int} \vee \mathsf{Bool}$. Null is a valid instantiation in this case, while Int is not a valid instantiation.

Expressions are extended with type application $e\,A$ and type abstraction $\Lambda(\alpha * G).e$. A type abstraction is also a value. Additionally, context Γ now also contains type variables with their respective disjointness constraints. The disjointness constraint of type variables is restricted to ground types (G), which includes all the types except type variables. Ground types are shown at the top left of Figure 9.

Subtyping, Typing and Operational Semantics. Figure 9 shows additional rules in the formalization of λ_u with polymorphism. Note that subtyping, typing, and reduction relations now have two contexts Δ and Γ. Subtyping is extended for the two newly added types. The subtyping rule for type variables is a special case of reflexivity (rule POLYS-TVAR)). Rule POLYS-ALL is interesting. It says that input and output types of two disjoint quantifiers are covariant in the subtype relation. This contrasts with calculi with bounded quantification and disjoint polymorphism [4], where the subtyping between the type bounds of the constraints is contravariant, and the subtyping between the types in the universal quantification body is covariant. Note that in the calculus that we formalized in Coq, we study parametric polymorphism without distributive subtyping rules.

Similarly, typing is extended to assign the type to two newly added expressions. Rule PTYP-TAP is for type applications and rule PTYP-TABS is for type abstractions. Rule POLYSTEP-TAPPL is standard reduction rule for type application. Rule POLYSTEP-TAPP replaces α with type B in expression e.

Disjointness. Disjointness has to be updated to accommodate type variables and disjoint quantifiers. The definition of algorithmic disjointness is roughly the same as discussed in Section 4, except that it takes an additional argument Γ. Context Γ is also an argument of LOS. LOS is extended to handle the additional cases of α and $\forall(\alpha * G).B$ and is shown at the top right of Figure 9. LOS returns $\forall(\alpha * \bot).\bot$ as the least ordinary subtype of $\forall(\alpha * G).B$. The type variable case is interesting. It returns the set difference of all ordinary subtypes and LOS of the disjointness constraint of type variable. Note that the disjointness constraint of type variables is restricted to ground types.

▶ **Definition 25** (Disjointness). $\Delta;\Gamma \vdash A * B ::= |A|_{\Delta;\Gamma} \cap |B|_{\Delta;\Gamma} = \{\}$.

Type-safety and Determinism. The extension with disjoint polymorphism retains the properties of type-soundness and determinism. All the metatheory is formalized in Coq theorem prover. Progress and determinism does not require significant changes for this extension. Type preservation requires the preservation of disjointness after substitution and disjointness narrowing along with disjointness weakening. Disjointness substitution states that if two types are disjoint before type substitution, they must be disjoint after type substitution as stated in Lemma 26. The disjointness narrowing relates disjointness and subtyping. It states that it is safe to change the bounds of type variables from subtypes to supertypes as stated in Lemma 27.

▶ **Lemma 26** (Disjointness Substitution). *If* $\Delta;\Gamma,\, \alpha * G_1 \vdash B * C$ *and* $\Delta;\Gamma \vdash G_2 * G_1$ *then* $\Delta;\Gamma[\alpha \rightsquigarrow G_2] \vdash B[\alpha \rightsquigarrow G_2] * C[\alpha \rightsquigarrow G_2]$

▶ **Lemma 27** (Disjointness Narrowing). *If* $\Delta; \Gamma, \alpha * G_1 \vdash B * C$ *and* $\Delta; \Gamma \vdash G_1 <: G_2$ *then* $\Delta; \Gamma, \alpha * G_2 \vdash B * C$

A.2 A More General Subtyping Rule for Bottom Types

As discussed in Section 4.3, Ceylon includes the following subtyping rule:

$$\frac{A * B}{A \wedge B <: \bot} \text{ S-DISJ}$$

It is possible to support, and in fact generalize, such a rule in λ_u. The idea is to employ our definition of lowest ordinary subtypes, and add the following rule to λ_u with intersection types:

$$\frac{|A| = \{\}}{A <: B} \text{ S-LOS}$$

Rule S-LOS is an interesting addition in subtyping of λ_u. It says that if the LOS returns the empty set for some type A, then A is a subtype of all types. In other words, such type behaves like a *bottom-like* type. Such rule generalizes the rule S-DISJ employed in Ceylon, since when A is an intersection type of two disjoint types, we get the empty set. Moreover, adding rule S-LOS makes rule S-BOT redundant as well, since the LOS for the bottom type is also the empty set. It is trivial to prove a lemma which says that $\bot$ is a subtype of all types. We drop rule S-BOT from the calculus discussed in Section 4 and prove Lemma 28 to show this property instead:

▶ **Lemma 28** (Bottom Type Least Subtype). $\bot <: A$.

A similar lemma can be proved to show that disjoint types are bottom-like (as in rule S-DISJ), when rule S-LOS is added to subtyping:

▶ **Lemma 29** (Disjont Intersections are Bottom-Like). *If* $A * B$ *then* $A \wedge B <: \bot$.

The use of rule S-LOS instead of rule S-DISJ also has the advantage that it does not create a mutual dependency between disjointness and subtyping. We can have the definition of disjointness, which depends only on subtyping and ordinary types, and the definition of subtyping, which depends on LOS but not on disjointness.

We have formalized and proved all the metatheory, including type soundness, transitivity of subtyping, soundness and completeness of disjointness and determinism for a variant of λ_u with intersection types, nominal types, standard subtyping and rule S-LOS in Coq.

A.3 Implementation of Disjoint Switches

Ceylon code runs on the Java Virtual Machine (JVM). A Ceylon program compiles to JVM bytecode. The final bytecode to which a Ceylon program is compiled to erase annotations for types not supported in the JVM. In particular, union types such as String $\vee$ Null are erased into `Object`. Disjoint switches are implemented by type casts. For each branch there is an *instanceof* to test the type of the branch and select a particular branch. An implementation of the λ_u calculus could also use a similar approach for compilation. In essence the use of union types and disjoint switches provides an elegant alternative to type-unsafe idioms, based on *instanceof* tests, that are currently widely used by Java programmers, while keeping comparable runtime performance.

Fair Termination of Multiparty Sessions

Luca Ciccone
University of Torino, Italy

Francesco Dagnino
University of Genova, Italy

Luca Padovani
University of Torino, Italy

Abstract

There exists a broad family of multiparty sessions in which the progress of one session participant is not unconditional, but depends on the choices performed by other participants. These sessions fall outside the scope of currently available session type systems that guarantee progress. In this work we propose the first type system ensuring that well-typed multiparty sessions, including those exhibiting the aforementioned dependencies, fairly terminate. Fair termination is termination under a fairness assumption that disregards those interactions deemed unfair and therefore unrealistic. Fair termination, combined with the usual safety properties ensured within sessions, not only is desirable *per se*, but it entails progress and enables a compositional form of static analysis such that the well-typed composition of fairly terminating sessions results in a fairly terminating program.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Type structures; Theory of computation → Program analysis

Keywords and phrases Multiparty sessions, fair termination, fair subtyping, deadlock freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.26

Related Version *Full Version*: `https://arxiv.org/abs/2205.08786` [10]

Funding The second author was partially supported by the MUR project T-LADIES (PRIN 2020TL3X8X).

Acknowledgements We are grateful to the anonymous ECOOP reviewers for their thoughtful and useful comments that helped us improving form and content of the paper.

1 Introduction

Sessions [24, 25, 27] are private conversations among processes following a protocol specification called session type. The decomposition of a distributed program into sessions enables its modular static analysis and the enforcement of useful properties through a type system. Examples of such properties are *communication safety* (no message of the wrong type is ever exchanged), *protocol fidelity* (messages are exchanged in the order prescribed by session types) and *deadlock freedom* (the program keeps running unless all sessions have terminated). These are all instances of *safety properties*, implying that "nothing bad" happens. In general, one is also interested in reasoning and possibly enforcing *liveness properties*, those implying that "something good" happens [39]. Examples of liveness properties are *junk freedom* (every message is eventually received), *progress* (every non-terminated participant of a session eventually performs an action) and *termination* (every session eventually comes to an end).

An enduring limitation of current type systems for multiparty sessions is that *they ensure progress for any participant of a session only when such progress can be established independently of the choices performed by the other participants.* To illustrate the impact of this limitation, consider a session made of three participants named buyer, seller and carrier

© Luca Ciccone, Francesco Dagnino, and Luca Padovani;
licensed under Creative Commons License CC-BY 4.0

in which the buyer aims at purchasing an unspecified number of items from the seller and the seller relies on a carrier for delivering the purchased items to the buyer. The buyer behaves according to the session type S that satisfies the equation

$$S = \mathsf{seller}!\mathsf{add}.S + \mathsf{seller}!\mathsf{pay}.!\mathsf{end} \tag{1}$$

indicating that it either pays the seller or it adds an item to the shopping cart and then repeats the same behavior. In this session type, add and pay are messages targeted to the participant with role seller. In turn, the seller accepts add messages from the buyer until a pay message is received, at which point it instructs the carrier to ship the items. Thus, its behavior is described by the session type T that satisfies the equation

$$T = \mathsf{buyer}?\mathsf{add}.T + \mathsf{buyer}?\mathsf{pay}.\mathsf{carrier}!\mathsf{ship}.!\mathsf{end} \tag{2}$$

Finally, the carrier just waits for the ship message from the seller. So, its behavior is described by the session type

$$\mathsf{seller}?\mathsf{ship}.?\mathsf{end} \tag{3}$$

No available type system is able to guarantee progress for every participant of this multiparty session. What makes this session somewhat difficult to reason about is that *the progress of the carrier is not unconditional but depends on the choices performed by the buyer*: the carrier can make progress only if the buyer eventually pays the seller.

In this work we propose a type system that guarantees the *fair termination* of sessions, that is termination under a *fairness assumption*. The assumption we make is an instance of *relative fairness* [45] and can be roughly spelled out as follows:

If termination is always possible, then it is inevitable. (4)

The multiparty session sketched above terminates under this fairness assumption: since it is always possible for the buyer to pay the seller and terminate, in every fair execution of the session the buyer eventually pays the seller, even though we do not know (nor do we impose) an upper bound to the number of items that the buyer may add to the shopping cart. Simply, the non-terminating execution of the session in which the buyer keeps adding items to the shopping cart but never pays is assumed unrealistic and so it can be ignored insofar as termination is concerned.

The reader might wonder why we focus on fair termination instead of considering some fair version of progress. There are three reasons why we think that fair termination is overall more appropriate than just progress. First of all, ensuring that sessions (fairly) terminate is consistent with the usual interpretation of the word "session" as an activity that lasts for a *finite amount of time*, even when the maximum duration of the activity is not known *a priori*. Second, *fair termination implies progress* when it is guaranteed along with the usual safety properties of sessions. Indeed, if the session eventually terminates, it must be the case that any non-terminated participant (think of the carrier waiting for a ship message) is guaranteed to eventually make progress, even when such progress *depends* on choices made by other participants (like the buyer sending pay to the seller). Last but not least, *fair session termination enables compositional reasoning* in the presence of multiple sessions. This is not true for progress: if an action on a session s is blocked by actions on a different session t, then knowing that the session t enjoys progress does not necessarily guarantee that the action on s will eventually be performed (the interaction on t might continue forever). On the contrary, knowing that t fairly terminates guarantees that the action on s will eventually be scheduled and performed, so that s may in turn progress towards termination.

Remarkably, the fairness assumption alone does not suffice to turn any multiparty session type system into one that ensures fair termination. In fact, there are several sources of potentially non-terminating behaviors that must be ruled out in well-typed processes:

1. Fairly terminating (and even finite) sessions may be chained, nested, interleaved in such a way that some pending activities are postponed forever. To avoid this problem, our type system makes sure that the effort required by a well-typed process in order to terminate remains finite. At the same time, it does not (always) prevent the modeling of processes that create an unbounded number of sessions.
2. The type-level constraints usually imposed to well-typed sessions – *duality* [24, 25, 27], *liveness* [46], *coherence* [9], just to mention a few – are in general too weak to entail fair session termination. Our type system adopts a stronger notion of "correct multiparty session" that entails fair termination. Variants of this notion have already appeared in the literature [5, 42], but we use it here for the first time to relate types and processes.
3. A certain mismatch is usually allowed between the structure of session types and the structure of the processes that adhere to those types. This mismatch is formalized by a subtyping relation for session types which, in its standard formulation [23], may introduce non-terminating behaviors. Our type system adopts *fair subtyping* [42], a liveness-preserving refinement of the standard subtyping relation for session types [23].

Summary of contributions. We present the first type system ensuring the fair termination of multiparty sessions and capable of addressing a number of natural communication patterns that are out of scope of existing multiparty session type systems [46, 48]. We exploit the compositional reasoning enabled by fair termination to prove a strong soundness result whereby a well-typed composition of fairly terminating sessions is a fairly terminating program (Theorem 5.4). This result scales smoothly also in presence of session chaining, session nesting, session interleaving, session delegation and dynamic session creation. In sharp contrast, the liveness properties ensured by previous multiparty session type systems are either limited to single-session programs [46, 48] or require a richer type structure [43, 15]. Our contributions extend and generalize previous work on the fair termination of binary sessions [14] and allow for the modeling of (intra-session) cyclic network topologies and of multiparty sessions that cannot be decomposed into equivalent (well-typed) binary sessions. Decidability of type checking is not substantially more difficult than the same problem in the binary setting [14]. *En passant*, in this paper we also provide a new characterization of fair subtyping for (multiparty) session types (Table 5) that is substantially simpler than those appearing in previous works [40, 42, 13, 14].

Structure of the paper. We recall the key notions related to fair termination (Section 2) before presenting our language of multiparty sessions (Section 3). Then, we define multiparty session types and fair subtyping (Section 4) and present the typing rules and the soundness properties of the type system (Section 5). In the latter part of the paper we illustrate a few more advanced examples of well-typed processes (Section 6), we discuss related work in more detail (Section 7) and we provide hints at further developments (Section 8).

2 Fair Termination

Since the notion of fair termination will apply to several different entities (session types, multiparty sessions, processes) here we define it for a generic reduction system. Later on we will show various instantiations of this definition. A *reduction system* is a pair $(\mathcal{S}, \rightarrow)$ where

$\mathcal{S}$ is a set of *states* and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a *reduction relation*. We adopt the following notation: we let C and D range over states; we write $C \rightarrow$ if there exists $D \in \mathcal{S}$ such that $C \rightarrow D$; we write $C \not\rightarrow$ if not $C \rightarrow$; we write $\Rightarrow$ for the reflexive, transitive closure of $\rightarrow$. We say that D is *reachable* from C if $C \Rightarrow D$.

As an example, the reduction system $(\{A, B\}, \{(A, A), (A, B)\})$ models an entity that can be in two states, A or B, and such that the entity may perform a reduction to remain in state A or a reduction to move from state A to state B. To formalize the evolution of an entity from a particular state we define *runs*.

▶ **Definition 2.1** (runs and maximal runs). *A* run *of C is a (finite or infinite) sequence $C_0 C_1 \dots C_i \dots$ of states such that $C_0 = C$ and $C_i \rightarrow C_{i+1}$ for every valid i. A run is* maximal *if either it is infinite or if its last state C_n is such that $C_n \not\rightarrow$.*

Hereafter we let ρ range over runs. Each run in the previously defined reduction system is either of the form A^n – a finite sequence of A – or of the form $A^n B$ – a finite sequence of A followed by one B – or A^ω – an infinite sequence of A. Among these, the runs of the form $A^n B$ and A^ω are maximal, whereas no run of the form A^n is maximal.

We now use runs to define different termination properties of states: we say that C is *weakly terminating* if there exists a maximal run of C that is finite; we say that C is *terminating* if every maximal run of C is finite; we say that C is *diverging* if every maximal run of C is infinite. *Fair termination* [21] is a termination property that only considers a subset of all (maximal) runs of a state, those that are considered to be "realistic" or "fair" according to some fairness assumption. The assumption that we make in this work, and that we stated in words in (4), is formalized thus:

▶ **Definition 2.2** (fair run). *A run is* fair *if it contains finitely many weakly terminating states. Conversely, a run is* unfair *if it contains infinitely many weakly terminating states.*

Continuing with the previous example, the runs of the form A^n and $A^n B$ are fair, whereas the run A^ω is unfair. In general, an unfair run is an execution in which termination is always within reach, but is never reached.

A key requirement of any fairness assumption is that it must be possible to extend every finite run to a maximal fair one. This property is called *feasibility* [4, 47] or *machine closure* [37]. It is easy to see that our fairness assumption is feasible:

▶ **Lemma 2.3.** *If ρ is a finite run, then there exists ρ' such that $\rho\rho'$ is a maximal fair run.*

Fair termination is finiteness of all maximal fair runs:

▶ **Definition 2.4** (fair termination). *We say that C is* fairly terminating *if every maximal fair run of C is finite.*

In the reduction system given above, A is fairly terminating. Indeed, all the maximal runs of the form $A^n B$ are finite whereas A^ω, which is the only infinite fair run of A, is unfair.

For the particular fairness assumption that we make, it is possible to provide a sound and complete characterization of fair termination that does not mention fair runs. This characterization will be useful to relate fair termination with the notion of correct multiparty session (Definition 4.2) and the soundness property of the type system (Theorem 5.4).

▶ **Theorem 2.5.** *Let $(\mathcal{S}, \rightarrow)$ be a reduction system and $C \in \mathcal{S}$. Then C is fairly terminating if and only if every state reachable from C is weakly terminating.*

Table 1 Syntax of processes.

P, Q, R	$::=$		**Process**			
		done	termination	$\mid$	$A\langle\overline{u}\rangle$	invocation
	$\mid$	$\mathsf{wait}\,u.P$	signal input	$\mid$	$\mathsf{close}\,u$	signal output
	$\mid$	$u[\mathsf{p}]?(x).P$	channel input	$\mid$	$u[\mathsf{p}]!v.P$	channel output
	$\mid$	$u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}$	tag input/output	$\mid$	$P \oplus Q$	choice
	$\mid$	$(s)(P_1 \mid \cdots \mid P_n)$	session	$\mid$	$\lceil u \rceil P$	cast

▶ Remark 2.6 (fair reachability of predicates [45]). Most fairness assumptions have the form "if *something* is infinitely often possible then *something* happens infinitely often" and, in this respect, our formulation of fair run (Definition 2.2) looks slightly unconventional. However, it is not difficult to realize that Definition 2.2 is an instance of the notion of fair reachability of predicates as defined by Queille and Sifakis [45, Definition 3]. According to Queille and Sifakis, a run ρ is fair with respect to some predicate $\mathcal{C} \subseteq \mathcal{S}$ if, whenever in ρ there are infinitely many states from which a state in $\mathcal{C}$ is reachable, then in ρ there are infinitely many occurrences of states in $\mathcal{C}$. When we take $\mathcal{C}$ to be $\not\rightarrow$, that is the set of terminated states that do not reduce, pretending that irreducible states should occur infinitely often in the run is nonsensical. So, the fairness assumption boils down to assuming that such states should *not* be reachable infinitely often, which is precisely the formulation of Definition 2.2. ⌟

3 A Calculus of Multiparty Sessions

In this section we define the calculus for multiparty sessions on which we apply our static analysis technique. The calculus is an extension of the one presented by Ciccone and Padovani [14] to multiparty sessions in the style of Scalas and Yoshida [46].

We use an infinite set of *variables* ranged over by x, y, z, an infinite set of *session names* ranged over by s and t, a set of *roles* ranged over by p, q, r, a set of *message tags* ranged over by m, and a set of *process names* ranged over by A, B, C. In the literature of sessions tags are usually called labels. We adopt a different terminology to avoid confusion with another notion of label that we introduce in Section 4. We use roles to distinguish the participants of a session. In particular, an *endpoint* $s[\mathsf{p}]$ consists of a session name s and a role p and is used by the participant with role p to interact with the other participants of the session s. We use u and v to range over *channels*, which are either variables or session endpoints. We write $\overline{x}$ and $\overline{u}$ to denote possibly empty sequences of variables and channels, extending this notation to other entities. We use π to range over the elements of the set $\{?, !\}$ of *polarities*, distinguishing input actions (?) from output actions (!).

A *program* is a finite set of *definitions* of the form $A(\overline{x}) \triangleq P$, at most one for each process name, where P is a term generated by the syntax shown in Table 1. The term done denotes the terminated process that performs no action. The term $A\langle\overline{u}\rangle$ denotes the invocation of the process with name A passing the channels $\overline{u}$ as arguments. When $\overline{u}$ is empty we just write A instead of $A\langle\rangle$. The term $\mathsf{close}\,u$ denotes the process that sends a termination signal on the channel u, whereas $\mathsf{wait}\,u.P$ denotes the process that waits for a termination signal from channel u and then continues as P. The term $u[\mathsf{p}]!v.P$ denotes the process that sends the channel v on the channel u to the role p and then continues as P. Dually, $u[\mathsf{p}]?(x).P$ denotes the process that receives a channel from the role p on the channel u and then continues as P where x is replaced with the received channel. The term $u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}$ denotes a process that exchanges one of the tags m_i on the channel u with the role p and then continues as P_i.

Table 2 Structural precongruence of processes.

[S-PAR-COMM]	$(s)(\overline{P} \mid P \mid Q \mid \overline{Q}) \preccurlyeq (s)(\overline{P} \mid Q \mid P \mid \overline{Q})$	
[S-PAR-ASSOC]	$(s)(\overline{P} \mid (t)(R \mid \overline{Q})) \preccurlyeq (t)((s)(\overline{P} \mid R) \mid \overline{Q})$	if $s \in \mathsf{fn}(R)$
[S-CAST-COMM]	$\lceil u \rceil \lceil v \rceil P \preccurlyeq \lceil v \rceil \lceil u \rceil P$	
[S-CAST-NEW]	$(s)(\lceil s[\mathsf{p}] \rceil P \mid \overline{Q}) \preccurlyeq (s)(P \mid \overline{Q})$	
[S-CAST-SWAP]	$(s)(\lceil t[\mathsf{p}] \rceil P \mid \overline{Q}) \preccurlyeq \lceil t[\mathsf{p}] \rceil (s)(P \mid \overline{Q})$	if $s \neq t$
[S-CALL]	$A\langle \overline{u} \rangle \preccurlyeq P\{\overline{u}/\overline{x}\}$	if $A(\overline{x}) \triangleq P$

Whether the tag is sent or received depends on the polarity π and, as it will be clear from the operational semantics, the polarity π also determines whether the process behaves as an internal choice (when π is !) or an external choice (when π is ?). In the first case the process chooses *actively* the tag being sent, whereas in the second case the process reacts *passively* to the tag being received. We assume that I is finite and non-empty and also that the tags m_i are pairwise distinct. For brevity, we write $u[\mathsf{p}]\pi\mathsf{m}_k.P_k$ instead of $u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}$ when I is the singleton set $\{k\}$. The term $P \oplus Q$ denotes a process that non-deterministically behaves either as P or as Q.

A term $(s)(P_1 \mid \cdots \mid P_n)$ with $n \geq 1$ denotes the parallel composition of n processes, each of them being a participant of the session s. Each process is associated with a distinct a role p_i and communicates in s through the endpoint $s[\mathsf{p}_i]$. Combining session creation and parallel composition in a single form is common in session type systems based on linear logic [6, 49, 38] and helps guaranteeing deadlock freedom. Finally, a *cast* $\lceil u \rceil P$ denotes a process that behaves exactly as P. This form is only relevant for the type system (Section 5) and denotes the fact that the type of u is subject to an application of subtyping.

The free and bound names of a process are defined as usual, the latter ones being easily recognizable as they occur within round parenteses. We write $\mathsf{fn}(P)$ for the set of free names of P and we identify processes modulo renaming of bound names. Note that $\mathsf{fn}(P)$ may contain variables and session names, but not endpoints. Occasionally we write $A(\overline{x}) \triangleq P$ as a predicate or side condition, meaning that P is the process associated with the process name A. For each of such definitions we assume that $\mathsf{fn}(P) \subseteq \{\overline{x}\}$.

The operational semantics of processes is given by the structural precongruence relation $\preccurlyeq$ defined in Table 2 and the reduction relation $\rightarrow$ defined in Table 3. As usual, structural precongruence allows us to rearrange the structure of processes without altering their meaning, whereas reduction expresses an actual computation or interaction step. The adoption of a structural *pre*congruence (as opposed to a more common congruence relation) is not strictly necessary, but it simplifies the technical development by reducing the number of cases we have to consider in proofs without affecting the properties of the calculus in any way.

Rules [S-PAR-COMM] and [S-PAR-ASSOC] state commutativity and associativity of parallel composition of processes (we write $\overline{P}$ to denote possibly empty parallel compositions of processes). In [S-PAR-ASSOC], the side condition $s \in \mathsf{fn}(R)$ makes sure that R is indeed a participant of the session s. Note that this rule only states right-to-left associativity. Left-to-right associativity is derivable from this rule and repeated uses of [S-PAR-COMM]. Rule [S-CAST-COMM] allows us to swap two consecutive casts. Rule [S-CAST-NEW] removes an unguarded cast on an endpoint of the restricted session (we refer to this operation as "performing the cast"). Rule [S-CAST-SWAP] swaps a cast and a restricted session as long as the endpoint in the cast refers to a different session. Finally, rule [S-CALL] unfolds a process invocation to its definition. Hereafter, we write $\{u/x\}$ for the capture-avoiding substitution of each free occurrence of x with u and $\{\overline{u}/\overline{x}\}$ for its natural extension to equal-length tuples

Table 3 Reduction of processes.

$$\frac{}{P_1 \oplus P_2 \to P_k}\ k \in \{1,2\}\ \text{[R-CHOICE]}$$

$$\frac{}{(s)(\mathsf{wait}\ s[\mathsf{p}].P \mid \mathsf{close}\ s[\mathsf{q}_1] \mid \cdots \mid \mathsf{close}\ s[\mathsf{q}_n]) \to P}\ \text{[R-SIGNAL]}$$

$$\frac{}{(s)(s[\mathsf{p}][\mathsf{q}]!v.P \mid s[\mathsf{q}][\mathsf{p}]?(x).Q \mid \overline{R}) \to (s)(P \mid Q\{v/x\} \mid \overline{R})}\ \text{[R-CHANNEL]}$$

$$\frac{}{(s)(s[\mathsf{p}][\mathsf{q}]!\{\mathsf{m}_i.P_i\}_{i\in I} \mid \overline{Q}) \to (s)(s[\mathsf{p}][\mathsf{q}]!\mathsf{m}_k.P_k \mid \overline{Q})}\ k \in I\ \text{[R-PICK]}$$

$$\frac{}{(s)(s[\mathsf{p}][\mathsf{q}]!\mathsf{m}_k.P \mid s[\mathsf{q}][\mathsf{p}]?\{\mathsf{m}_i.Q_i\}_{i\in I} \mid \overline{R}) \to (s)(P \mid Q_k \mid \overline{R})}\ k \in I\ \text{[R-TAG]}$$

$$\frac{P \to Q}{(s)(P \mid \overline{R}) \to (s)(Q \mid \overline{R})}\ \text{[R-PAR]} \qquad \frac{P \to Q}{\lceil u \rceil P \to \lceil u \rceil Q}\ \text{[R-CAST]} \qquad \frac{P \preccurlyeq P' \quad P' \to Q' \quad Q' \preccurlyeq Q}{P \to Q}\ \text{[R-STRUCT]}$$

of variables and names. The rules [S-CAST-NEW], [S-CAST-SWAP] and [S-CALL] are not invertible: by [S-CAST-NEW] casts can only be removed but never added; by [S-CAST-SWAP] casts can only be moved closer to their restriction, so that they can be eventually performed by [S-CAST-NEW]; by [S-CALL] process invocations can only be unfolded.

The reduction relation is quite standard. Rule [R-CHOICE] reduces $P_1 \oplus P_2$ to either P_1 or P_2, non deterministically. Rule [R-SIGNAL] terminates a session in which all participants ($\mathsf{q}_1, \dots, \mathsf{q}_n$) but one ($\mathsf{p}$) are sending a termination signal and p is waiting for it; the resulting process is the continuation of the participant p. Rule [R-CHANNEL] models the exchange of a channel among two participants of a session. Rule [R-PICK] models an internal choice whereby a process picks one particular tag m_k to send on a session. Rule [R-TAG] synchronizes two participants p and q on the tag chosen by p. Finally, rules [R-PAR], [R-CAST] and [R-STRUCT] close reductions under parallel compositions and casts and by structural precongruence.

In the rest of this section we illustrate the main features of the calculus with some examples. For none of them the existing multiparty session type systems are able to guarantee progress.

▶ **Example 3.1** (purchase). We model a particular instance of the buyer-seller-carrier interaction that we have informally discussed in Section 1 with the following definitions:

$$\begin{aligned}
\mathit{Main} &\triangleq (s)(\mathit{Buyer}\langle s[\mathsf{buyer}]\rangle \mid \mathit{Seller}\langle s[\mathsf{seller}]\rangle \mid \mathit{Carrier}\langle s[\mathsf{carrier}]\rangle)\\
\mathit{Buyer}(x) &\triangleq x[\mathsf{seller}]!\{\mathsf{add}.x[\mathsf{seller}]!\mathsf{add}.\mathit{Buyer}\langle x\rangle, \mathsf{pay}.\mathsf{close}\ x\}\\
\mathit{Seller}(x) &\triangleq x[\mathsf{buyer}]?\{\mathsf{add}.\mathit{Seller}\langle x\rangle, \mathsf{pay}.x[\mathsf{carrier}]!\mathsf{ship}.\mathsf{close}\ x\}\\
\mathit{Carrier}(x) &\triangleq x[\mathsf{seller}]?\mathsf{ship}.\mathsf{wait}\ x.\mathsf{done}
\end{aligned}$$

Note that the buyer either sends pay or it sends two add messages in a row before repeating this behavior. That is, this particular buyer always adds an even number of items to the shopping cart. Nonetheless, the buyer periodically has a chance to send a pay message and terminate. Therefore, the execution of the program in which the buyer only sends add is unfair according to Definition 2.2 hence this program is fairly terminating. ⌟

▶ **Example 3.2** (purchase with negotiation). Consider a variation of Example 3.1 in which the buyer, before making the payment, negotiates with a secondary buyer for an arbitrarily long time. The interaction happens in two nested sessions, an outer one involving the primary buyer, the seller and the carrier, and an inner one involving only the two buyers. We model the interaction as the program below, in which we collapse role names to their initials.

$$\begin{aligned}
\mathit{Main} &\triangleq (s)(\mathit{Buyer}\langle s[\mathsf{b}]\rangle \mid \mathit{Seller}\langle s[\mathsf{s}]\rangle \mid \mathit{Carrier}\langle s[\mathsf{c}]\rangle)\\
\mathit{Buyer}(x) &\triangleq x[\mathsf{s}]!\mathsf{query}.x[\mathsf{s}]?\mathsf{price}.(t)(\mathit{Buyer}_1\langle x, t[\mathsf{b}_1]\rangle \mid \mathit{Buyer}_2\langle t[\mathsf{b}_2]\rangle)\\
\mathit{Seller}(x) &\triangleq x[\mathsf{b}]?\mathsf{query}.x[\mathsf{b}]!\mathsf{price}.x[\mathsf{b}]?\{\mathsf{pay}.x[\mathsf{c}]!\mathsf{ship}.\mathsf{close}\,x, \mathsf{cancel}.x[\mathsf{c}]!\mathsf{cancel}.\mathsf{close}\,x\}\\
\mathit{Carrier}(x) &\triangleq x[\mathsf{s}]?\{\mathsf{ship}.x[\mathsf{b}]!\mathsf{box}.\mathsf{close}\,x, \mathsf{cancel}.\mathsf{close}\,x\}\\
\mathit{Buyer}_1(x,y) &\triangleq y[\mathsf{b}_2]!\{\mathsf{split}.y[\mathsf{b}_2]?\{\mathsf{yes}.\lceil x\rceil x[\mathsf{s}]!\mathsf{ok}.x[\mathsf{c}]?\mathsf{box}.\mathsf{wait}\,x.\mathsf{wait}\,y.\mathsf{done},\\
&\qquad\qquad\qquad\qquad \mathsf{no}.\mathit{Buyer}_1\langle x,y\rangle\},\\
&\qquad\qquad \mathsf{giveup}.\mathsf{wait}\,y.\lceil x\rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\,x.\mathsf{done}\}\\
\mathit{Buyer}_2(y) &\triangleq y[\mathsf{b}_1]?\{\mathsf{split}.y[\mathsf{b}_1]!\{\mathsf{yes}.\mathsf{close}\,y, \mathsf{no}.\mathit{Buyer}_2\langle y\rangle\}, \mathsf{giveup}.\mathsf{close}\,y\}
\end{aligned}$$

The buyer queries the seller which replies with a price. At this point, *Buyer* creates a new session t and forks as a primary buyer Buyer_1 and a secondary buyer Buyer_2. The interaction between the two sub-buyers goes on until either Buyer_1 gives up or Buyer_2 accepts its share of the price. In the former case, the primary buyer waits for the internal session to terminate and cancels the order with the seller which, in turn, aborts the transaction with the carrier. In the latter case, the buyer confirms the order to the seller, which then instructs the carrier to ship a box to the buyer.

Note that the outermost session s, taken in isolation, terminates in a bounded number of interactions, but its progress cannot be established without assuming that the innermost session t terminates. In particular, if the two buyers keep negotiating forever, the seller and the carrier starve. However, the innermost session can terminate if Buyer_1 sends giveup to Buyer_2 or if Buyer_2 sends yes to Buyer_1. Thus, the run in which the two buyers negotiate forever is unfair, the session t fairly terminates and the session s terminates as well.

On the technical side, note that the definition of Buyer_1 contains two casts on the variable x. As we will see in Example 6.1, these casts are necessary for the typeability of Buyer_1 to account for the fact that x is used *differently* in two distinct branches of the process. ⌟

▶ **Example 3.3** (parallel merge sort). To illustrate an example of program that creates an unbounded number of sessions we model a parallel version of the merge sort algorithm.

$$\begin{aligned}
\mathit{Main} &\triangleq (s)(s[\mathsf{m}][\mathsf{w}]!\mathsf{req}.s[\mathsf{m}][\mathsf{w}]?\mathsf{res}.\mathsf{wait}\,s.\mathsf{done} \mid \mathit{Sort}\langle s[\mathsf{w}]\rangle)\\
\mathit{Sort}(x) &\triangleq x[\mathsf{m}]?\mathsf{req}.((t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_2]\rangle) \oplus x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\,x)\\
\mathit{Merge}(x,y) &\triangleq y[\mathsf{w}_1]!\mathsf{req}.y[\mathsf{w}_2]!\mathsf{req}.y[\mathsf{w}_1]?\mathsf{res}.y[\mathsf{w}_2]?\mathsf{res}.\mathsf{wait}\,y.x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\,x
\end{aligned}$$

The program starts as a single session s in which a master m sends the initial collection of data to the worker w as a req message and waits for the result. The worker is modeled as a process *Sort* that decides whether to sort the data by itself (right branch of the choice in *Sort*), in which case it sends the result directly to the master, or to partition the collection (left branch of the choice in *Sort*). In the latter case, it creates a new session t in which it sends requests to two sub-workers w_1 and w_2, it gathers the partial results from them and gets back to the master with the complete result.

Since a worker may always choose to start two sub-workers in a new session, the number of sessions that may be created by this program is unbounded. At the same time, each worker may also choose to complete its task without creating new sessions. So, while in principle there exists a run of this program that keeps creating new sessions forever, this run is unfair according to Definition 2.2. ⌟

4 Multiparty Session Types and Fair Subtyping

In this section we define syntax and semantics of multiparty session types (Section 4.1) as well as an inference system for fair subtyping (Section 4.2).

4.1 Syntax and Semantics

A *session type* is a regular tree [16] coinductively generated by the productions below:

Session type $S, T, U, V ::= \pi\mathsf{end} \mid \sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i \mid \mathsf{p}\pi S.T$

The session type $\pi\mathsf{end}$ describes the behavior of a process that sends/receives a termination signal. The session type $\sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i$ describes the behavior of a process that sends to or receives from the participant p one of the tags m_i and then behaves according to S_i. Note that the source or destination role p and the polarity π are the same in every branch. We require that I is not empty and $i, j \in I$ with $i \neq j$ implies $\mathsf{m}_i \neq \mathsf{m}_j$. Occasionally we write $\mathsf{p}\pi\mathsf{m}_1.S_1 + \cdots + \mathsf{p}\pi\mathsf{m}_n.S_n$ instead of $\sum_{i=1}^{n} \mathsf{p}\pi\mathsf{m}_i.S_i$. Finally, a session type $\mathsf{p}\pi S.T$ describes the behavior of a process that sends to or receives from the participant p an endpoint of type S and then behaves according to T. We often specify infinite session types as solutions of equations of the form $S = \cdots$ where the metavariable S may occur on the right hand side of $=$ guarded by at least one prefix. A regular tree satisfying such equation is guaranteed to exist and to be unique [16].

In order to describe a whole multiparty session at the level of types we introduce the notion of *session map*.

▶ **Definition 4.1** (session map). *A* session map *is a finite, partial map from roles to session types written $\{\mathsf{p}_i \triangleright S_i\}_{i\in I}$. We let M and N range over session maps, we write* $\mathsf{dom}(M)$ *for the domain of M, we write $M \mid N$ for the union of M and N when* $\mathsf{dom}(M) \cap \mathsf{dom}(N) = \emptyset$*, and we abbreviate the singleton map $\{\mathsf{p} \triangleright S\}$ as $\mathsf{p} \triangleright S$.*

We describe the evolution of a session at the level of types by means of a *labeled transition system* for session maps. Labels are generated by the grammar below:

Label $\ell ::= \tau \mid \alpha$ **Action** $\alpha, \beta ::= \pi\checkmark \mid \mathsf{p} \triangleright \mathsf{q}\pi\mathsf{m} \mid \mathsf{p} \triangleright \mathsf{q}\pi S$

The label τ represents either an internal action performed by a participant independently of the others or a synchronization between two participants. The labels of the form $\pi\checkmark$ describe the input/output of termination signals, whereas the labels of the form $\mathsf{p} \triangleright \mathsf{q}\pi\mathsf{m}$ and $\mathsf{p} \triangleright \mathsf{q}\pi S$ represent the input/output of a tag m or of an endpoint of type S.

The labeled transition system is defined by the rules in Table 4, most of which are straightforward. Rule [L-PICK] models the fact that the participant p may internally choose one particular tag m_k before sending it to q. The chosen tag is not negotiable with the receiver. Rule [L-TERMINATE] models termination of a session. A session terminates when there is exactly one participant waiting for the termination signal and all the others are sending it. This property follows from a straightforward induction on the derivation of $M \xrightarrow{?\checkmark} N$ using [L-TERMINATE] and [L-END]. The existence of a single participant waiting for the termination signal ensures that there is a uniquely determined continuation process after the session has been closed. Finally, rule [L-SYNC] models the synchronization between two participants performing complementary actions. The complement of an action α, denoted by $\overline{\alpha}$, is the partial operation defined by the equations

$$\overline{\mathsf{p} \triangleright \mathsf{q}\pi\mathsf{m}} \overset{\text{def}}{=} \mathsf{q} \triangleright \mathsf{p}\overline{\pi}\mathsf{m} \qquad \overline{\mathsf{p} \triangleright \mathsf{q}\pi S} \overset{\text{def}}{=} \mathsf{q} \triangleright \mathsf{p}\overline{\pi} S$$

Table 4 Labeled transition system for session maps.

$$\frac{}{\mathsf{p} \triangleright \pi\mathsf{end} \xrightarrow{\pi\checkmark} \mathsf{p} \triangleright \pi\mathsf{end}}\;[\text{L-END}] \qquad \frac{}{\mathsf{p} \triangleright \mathsf{q}\pi U.S \xrightarrow{\mathsf{p}\triangleright\mathsf{q}\pi U} \mathsf{p} \triangleright S}\;[\text{L-CHANNEL}]$$

$$\frac{}{\mathsf{p} \triangleright \sum_{i\in I} \mathsf{q}!\mathsf{m}_i.S_i \xrightarrow{\tau} \mathsf{p} \triangleright \mathsf{q}!\mathsf{m}_k.S_k}\; k \in I \;[\text{L-PICK}] \qquad \frac{}{\mathsf{p} \triangleright \sum_{i\in I} \mathsf{q}\pi\mathsf{m}_i.S_i \xrightarrow{\mathsf{p}\triangleright\mathsf{q}\pi\mathsf{m}_k} \mathsf{p} \triangleright S_k}\; k \in I \;[\text{L-TAG}]$$

$$\frac{M \xrightarrow{\tau} M'}{M \mid N \xrightarrow{\tau} M' \mid N}\;[\text{L-TAU}] \qquad \frac{M \xrightarrow{?\checkmark} M' \qquad N \xrightarrow{!\checkmark} N'}{M \mid N \xrightarrow{?\checkmark} M' \mid N'}\;[\text{L-TERMINATE}] \qquad \frac{M \xrightarrow{\overline{\alpha}} M' \qquad N \xrightarrow{\alpha} N'}{M \mid N \xrightarrow{\tau} M' \mid N'}\;[\text{L-SYNC}]$$

where $\overline{\pi}$ denotes the complement of the polarity π. The complement of actions of the form $\pi\checkmark$ is undefined, so rule [L-SYNC] cannot be applied to terminated sessions. Hereafter we write $\Rightarrow$ for the reflexive, transitive closure of $\xrightarrow{\tau}$ and $\overset{\alpha}{\Longrightarrow}$ for the composition $\Rightarrow\xrightarrow{\alpha}$.

We call *coherence* the property of multiparty sessions that we wish to enforce with our type system, namely the fact that a session can always terminate no matter how it evolves. We formulate coherence directly on the transition system of session maps, in line with the approach of Scalas and Yoshida [46] and without introducing global types.

▶ **Definition 4.2.** *We say that M is* coherent*, notation $\#M$, if $M \Rightarrow N$ implies $N \overset{?\checkmark}{\Longrightarrow}$.*

The term "coherence" is borrowed from Carbone et al. [8, 9], although the property is actually stronger than the one of Carbone et al. as it entails fair termination of multiparty sessions through Theorem 2.5. In particular, if we consider the reduction system whose states are session maps and whose reduction relation is $\xrightarrow{\tau}$, then $\#M$ implies M fairly terminating.

▶ **Example 4.3** (buyer-seller-carrier session map)**.** Consider the session types

$$S_b = \mathsf{seller}!\mathsf{add}.\mathsf{seller}!\mathsf{add}.S_b + \mathsf{seller}!\mathsf{pay}.!\mathsf{end}$$
$$S_s = \mathsf{buyer}?\mathsf{add}.S_s + \mathsf{buyer}?\mathsf{pay}.\mathsf{carrier}!\mathsf{ship}.!\mathsf{end}$$
$$S_c = \mathsf{seller}?\mathsf{ship}.?\mathsf{end}$$

which describe the behavior of the processes *Buyer*, *Seller* and *Carrier* in Example 3.1. The session map $\mathsf{buyer} \triangleright S_b \mid \mathsf{seller} \triangleright S_s \mid \mathsf{carrier} \triangleright S_c$ is coherent. To see that, consider any interaction between the buyer and the seller. One of two cases applies: either the buyer has sent an even number of add messages to the seller, in which case it can send pay and the session eventually terminates, or the buyer has sent an odd number of add messages to the seller, in which case it can send one more add message followed by a pay message and once again the session eventually terminates. ⌟

Coherence allows us to provide a semantic definition of *fair subtyping*, the relation that defines the safe substitution principle for session endpoints in our type system.

▶ **Definition 4.4** (fair subtyping)**.** *We say that S is a* fair subtype *of T, notation $S \sqsubseteq T$, if $M \mid \mathsf{p} \triangleright S$ coherent implies $M \mid \mathsf{p} \triangleright T$ coherent for every M and p.*

Definition 4.4 does not say much about the properties of fair subtyping except for the fact that it is a coherence-preserving preorder. For this reason, we devote Section 4.2 to defining an alternative characterization of fair subtyping that highlights its relationship with the standard subtyping relation for session types [23].

Table 5 Inference system for fair subtyping.

$$\frac{}{\pi\mathsf{end} \leqslant_n \pi\mathsf{end}}\ [\text{F-END}] \qquad \frac{S \leqslant_n T}{\mathsf{p}\pi U.S \leqslant_n \mathsf{p}\pi U.T}\ [\text{F-CHANNEL}] \qquad \frac{\forall i \in I : S_i \leqslant_{n_i} T_i \qquad \forall i \in I : n_i \leq n}{\sum_{i\in I} \mathsf{p?m}_i.S_i \leqslant_n \sum_{i\in I\cup J} \mathsf{p?m}_i.T_i}\ [\text{F-TAG-IN}]$$

$$\frac{\forall i \in I : S_i \leqslant_{n_i} T_i \qquad \forall i \in I : n_i \leq n}{\sum_{i\in I} \mathsf{p!m}_i.S_i \leqslant_n \sum_{i\in I} \mathsf{p!m}_i.T_i}\ [\text{F-TAG-OUT-1}] \qquad \frac{\forall i \in I : S_i \leqslant_{n_i} T_i \qquad \exists i \in I : n_i < n}{\sum_{i\in I\cup J} \mathsf{p!m}_i.S_i \leqslant_n \sum_{i\in I} \mathsf{p!m}_i.T_i}\ [\text{F-TAG-OUT-2}]$$

4.2 Inference System for Fair Subtyping

Consider the relation $\leqslant_n$ coinductively defined by the inference system in Table 5, where n ranges over natural numbers. The characterization of fair subtyping that we consider is the relation $\leqslant \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} \leqslant_n$. The rules for deriving $S \leqslant_n T$ are quite similar to those of the standard subtyping relation for session types [23]: [F-END] states reflexivity of subtyping on terminated session types; [F-CHANNEL] relates higher-order session types with the same polarity and payload type; [F-TAG-IN] is the usual covariant rule for the input of tags (the set of tags in the larger session type includes those in the smaller one); [F-TAG-OUT-2] is the usual contravariant rule for the output of tags (the set of tags in the smaller session type includes those in the larger one). Overall, these rules entail a "simulation" between the behaviors described by S and T whereby all inputs offered by S are also offered by T and all outputs performed by T are also performed by S. The main differences between $\leqslant$ and the subtyping relation of Gay and Hole [23] are the presence of an invariant rule for outputs [F-TAG-OUT-1] and the natural number n annotating each subtyping judgment $S \leqslant_n T$. Intuitively, this number estimates how much S and T differ in terms of performed outputs. In all rules but [F-TAG-OUT-2], the annotation in the conclusion of the rule is just an upper bound of the annotations found in the premises. In [F-TAG-OUT-2], where the sets of output tags in related session types may differ, the annotation n is required to be a *strict* upper bound for at least one of the premises. That is, there must be at least one premise in which the annotation strictly decreases, while no restriction is imposed on the others. Intuitively, this ensures the existence of a tag shared by the two related session types whose corresponding continuations are slightly less different. So, the annotation n provides an upper bound to the number of applications of [F-TAG-OUT-2] along any path (i.e. any sequence of actions) shared by S and T that leads to termination. In the particular case when $n = 0$, the rule [F-TAG-OUT-2] cannot be applied, so that T may perform all the outputs also performed by S.

► **Example 4.5.** Consider the session type $S = \mathsf{seller!add}.S + \mathsf{seller!pay.!end}$, which describes the behavior of the buyer in Equation (1) purchasing an arbitrary number of items, $T = \mathsf{seller!add.seller!add}.T + \mathsf{seller!pay.!end}$, which describes the behavior of the buyer in Example 3.1 always purchasing an even number of items, and $U = \mathsf{seller!add}.U$, which describes the behavior of a buyer attempting to purchase an infinite number of items without ever paying the seller. We have $S \leqslant T$ and $S \not\leqslant U$. Indeed, we can derive

$$\dfrac{\dfrac{\dfrac{\vdots}{S \leqslant_1 T}}{S \leqslant_2 \mathsf{seller!add}.T}\ [\text{F-TAG-OUT-2}] \qquad \dfrac{}{\mathsf{!end} \leqslant_0 \mathsf{!end}}\ [\text{F-END}]}{S \leqslant_1 T}\ [\text{F-TAG-OUT-2}]$$

but there is no derivation for $S \leqslant_n U$ no matter how large n is chosen. Note that there are infinitely many sequences of actions of S that cannot be performed by both T and U. In particular, T cannot perform any sequence of actions consisting of an odd number of add outputs followed by a pay output, whereas U cannot perform any sequence of add outputs followed by a pay output. Nonetheless, there is a path shared by S and T that leads into a region of S and T in which no more differences are detectable. The annotations in the derivation tree measures the distance of each judgment from such region. In the case of S and U, there is no shared path that leads to a region where no differences are detectable. ⌟

▶ **Example 4.6.** Consider the session types $S = \mathsf{player}?\mathsf{play}.(\mathsf{player}!\mathsf{win}.S + \mathsf{player}!\mathsf{lose}.S) + \mathsf{player}?\mathsf{quit}.!\mathsf{end}$ and $T = \mathsf{player}?\mathsf{play}.\mathsf{player}!\mathsf{lose}.T + \mathsf{player}?\mathsf{quit}.!\mathsf{end}$ describing the behavior of two slot machines, an unbiased one in which the player may win at every play and a biased one in which the player never wins. If we try to build a derivation for $S \leqslant_n T$ we obtain

$$\dfrac{\dfrac{\dfrac{\vdots}{S \leqslant_{n-1} T}}{\mathsf{player}!\mathsf{win}.S + \mathsf{player}!\mathsf{lose}.S \leqslant_n \mathsf{player}!\mathsf{lose}.T}\ [\text{F-TAG-OUT-1}] \quad \dfrac{}{!\mathsf{end} \leqslant_n !\mathsf{end}}\ [\text{F-END}]}{S \leqslant_n T}\ [\text{F-TAG-IN}]$$

which would contain an infinite branch with strictly decreasing annotations. Therefore, we have $S \not\leqslant T$. In this case there exists a shared path leading into a region of S and T in which no more differences are detectable between the two protocols, but this path starts from an input. The fact that S is *not* a fair subtype of T has a semantic justification. Think of a player that deliberately insists on playing until it wins. This is possible when player interacts with the unbiased slot machine S but not with the biased one T. ⌟

In the rest of this section we study the fundamental properties of $\leqslant$, starting from the non-obvious fact that it is a preorder.

▶ **Theorem 4.7.** *$\leqslant$ is a preorder.*

While reflexivity of $\leqslant$ is trivial to prove (since [F-TAG-OUT-2] is never necessary, it suffices to only consider judgments with a 0 annotation), transitivity is surprisingly complex. The challenging part of proving that from $S \leqslant_m U$ and $U \leqslant_n T$ we can derive $S \leqslant_k T$ is to come up with a feasible annotation k. As it turns out, such k depends not only on m and n, but also on annotations found in different regions of the derivation trees that prove $S \leqslant_m U$ and $U \leqslant_n T$. In particular, the "difference" of S and T is not simply the "maximum difference" or "the sum of the differences" of S and U and of U and T. More in detail, we first show that we can always find a derivation of $S \leqslant_m U$ where the rank annotations of all judgements occurring in it are below some $h \geq m$; then, the judgement $S \leqslant_k T$ is provable for $k = m + (1+h)n$. For previous characterizations of fair subtyping [40, 42, 13, 14], transitivity has been established indirectly by relating the inference system of fair subtyping (Table 5) with its semantic definition (Definition 4.4). For Theorem 4.7 we are able to provide a direct proof [10].

Now we establish the connection between $\leqslant$ and $\sqsubseteq$ (Definition 4.4). First of all, we prove that $\leqslant$ is coherence-preserving just like $\sqsubseteq$ is.

▶ **Theorem 4.8** (soundness). *If $S \leqslant T$ then $S \sqsubseteq T$.*

The proof of this result relies on a key property of $\leqslant$ not enjoyed by the usual subtyping relation on session types [23]: when $S \leqslant T$ and $M \mid \mathsf{p} \triangleright S$ is coherent, the session map $M \mid \mathsf{p} \triangleright T$ can successfully terminate. The rank annotation on subtyping judgements is used to set up an appropriate inductive argument for proving this property.

Theorem 4.8 alone suffices to justify the adoption of $\leqslant$ as fair subtyping relation, but we are interested in understanding to which extent $\leqslant$ covers $\sqsubseteq$. In this respect, it is quite easy to see that there exist session types that are related by $\sqsubseteq$ but not by $\leqslant$. For example, consider $S = \mathsf{p}!\mathsf{a}.S$ and $T = \mathsf{p}?\mathsf{b}.T$ and observe that these two session types describe completely different protocols (the output of infinitely many a's in the case of S and the input of infinitely many b's in the case of T). In particular, we have $S \not\leqslant T$ and $T \not\leqslant S$ but also $S \sqsubseteq T$ and $T \sqsubseteq S$. That is, S and T are *unrelated* according to $\leqslant$ but they are *equivalent* according to $\sqsubseteq$. This equivalence is justified by the fact that there exists no coherent session map in which S and T could play any role, because none of them can ever terminate.

This discussion hints at the possibility that, if we restrict the attention to those session types that *can* terminate, which are the interesting ones as far as this work is concerned, then we can establish a tighter correspondence between $\leqslant$ and $\sqsubseteq$. We call such session types *bounded*, because they describe protocols for which termination is always within reach.

▶ **Definition 4.9** (bounded session type). *We say that a session type is* bounded *if all of its subtrees contain a* $\pi\mathsf{end}$ *leaf.*

Note that a *finite* session type is always bounded but not every bounded session type is finite. If we consider the reduction system in which states are session types and we have $S \to T$ if T is an immediate subtree of S, then S is bounded if and only if S is fairly terminating. Now, for the family of bounded session types we can prove a *relative completeness* result for $\leqslant$ with respect to $\sqsubseteq$.

▶ **Theorem 4.10** (relative completeness). *If S is bounded and $S \sqsubseteq T$ then $S \leqslant T$.*

The proof of Theorem 4.10 is done by contradiction. We show that, for any bounded S, if $S \leqslant T$ does not hold then we can build a session map M called *discriminator* such that $M \mid \mathsf{p} \triangleright S$ is coherent and $M \mid \mathsf{p} \triangleright T$ is not, which contraddicts the hypothesis $S \sqsubseteq T$. The boundedness of S is necessary to make sure that it is always possible to find a session map N such that $N \mid \mathsf{p} \triangleright S$ is coherent.

5 Type System

In this section we describe the type system for the calculus of multiparty sessions of Section 3. The typing judgments have the form $\Gamma \vdash^n P$, meaning that the process P is well typed in the typing context Γ and has rank n. As usual, the *typing context* is a map associating channels with session types and is meant to contain an association for each name in $\mathsf{fn}(P)$. We write $u_1 : S_1, \dots, u_n : S_n$ for the map with domain $\{u_1, \dots, u_n\}$ that associates u_i with S_i. Occationally we write $\overline{u : S}$ for the same context, when the number and the specific associations are unimportant. We also assume that endpoints occurring in a typing context have different session names. That is, $s[\mathsf{p}], s[\mathsf{q}] \in \mathsf{dom}(\Gamma)$ implies $\mathsf{p} = \mathsf{q}$. This constraint makes sure that each well-typed process plays exactly one role in each of the sessions in which it participates. It is also a common assumption made in all multiparty session calculi. We use Γ and Δ to range over typing contexts, we write $\emptyset$ for the empty context and Γ, Δ for the union of Γ and Δ when they have disjoint domains and disjoint sets of session names. The *rank* n in a typing judgment estimates the number of sessions that P has to create and the number of casts that P has to perform in order to terminate. The fact that the rank is finite suggests that so is the effort required by P to terminate.

The typing rules are shown in Table 6 as a *generalized inference system* [3, 17, 11, 18] in which, roughly speaking, the singly-lined rules are interpreted coinductively and the doubly-lined rules – called *corules* – are interpreted inductively. We will come back with a more

Table 6 Typing rules.

$$\text{[T-DONE]}\ \frac{}{\emptyset \vdash^n \mathsf{done}} \qquad \text{[T-CALL]}\ \frac{\overline{u : S} \vdash^n P\{\overline{u}/\overline{x}\}}{\overline{u : S} \vdash^{n+m} A\langle\overline{u}\rangle}\ A : [\overline{S}; n], A(\overline{x}) \triangleq P \qquad \text{[T-WAIT]}\ \frac{\Gamma \vdash^n P}{\Gamma, u : ?\mathsf{end} \vdash^n \mathsf{wait}\, u.P}$$

$$\text{[T-CLOSE]}\ \frac{}{u : !\mathsf{end} \vdash^n \mathsf{close}\, u} \qquad \text{[T-CHANNEL-IN]}\ \frac{\Gamma, u : T, x : S \vdash^n P}{\Gamma, u : \mathsf{p}?S.T \vdash^n u[\mathsf{p}]?(x).P} \qquad \text{[T-CHANNEL-OUT]}\ \frac{\Gamma, u : T \vdash^n P}{\Gamma, u : \mathsf{p}!S.T, v : S \vdash^n u[\mathsf{p}]!v.P}$$

$$\text{[T-TAG]}\ \frac{\forall i \in I : \Gamma, u : S_i \vdash^n P_i}{\Gamma, u : \sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i \vdash^n u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}} \qquad \text{[T-CHOICE]}\ \frac{\Gamma \vdash^{n_1} P_1 \qquad \Gamma \vdash^{n_2} P_2}{\Gamma \vdash^{n_k} P_1 \oplus P_2}\ k \in \{1, 2\}$$

$$\text{[T-CAST]}\ \frac{\Gamma, u : T \vdash^n P}{\Gamma, u : S \vdash^{m+n} \lceil u \rceil P}\ S \leqslant_m T \qquad \text{[T-PAR]}\ \frac{\forall i \in \{1, \dots, h\} : \Gamma_i, s[\mathsf{p}_i] : S_i \vdash^{n_i} P_i}{\Gamma_1, \dots, \Gamma_h \vdash^{1+n_1+\cdots+n_h} (s)(P_1 \mid \cdots \mid P_h)}\ \#\{\mathsf{p}_i \triangleright S_i\}_{i=1..h}$$

$$\text{[CO-TAG]}\ \frac{\Gamma, u : S_k \vdash^n P_k}{\Gamma, u : \sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i \vdash^n u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}}\ k \in I \qquad \text{[CO-CHOICE]}\ \frac{\Gamma \vdash^n P_k}{\Gamma \vdash^n P_1 \oplus P_2}\ k \in \{1, 2\}$$

detailed intuition later on (Definition 5.1), although we will not provide a formal definition of the interpretation of a generalized inference system in this paper. The interested reader may refer to the cited literature for details. We type check a program $\{A_i(\overline{x_i}) \triangleq P_i\}_{i\in I}$ under a global set of assignments $\{A_i : [\overline{S_i}; n_i]\}_{i\in I}$ associating each process name A_i with a tuple of session types $\overline{S_i}$, one for each of the variables in $\overline{x_i}$, and a rank n_i. The program is well typed if $\overline{x_i : S_i} \vdash^{n_i} P_i$ is derivable for every $i \in I$, establishing that the tuple $\overline{S_i}$ corresponds to the way the variables $\overline{x_i}$ are used by P_i and that n_i is a feasible rank annotation for P_i. We now describe the typing rules in detail.

The rule [T-DONE] states that the terminated process is well typed in the empty context, to make sure that no unused channels are left behind. Note that done can be given any rank, since it performs no casts and it creates no new sessions. The rule [T-CALL] checks that a process invocation $A\langle\overline{u}\rangle$ is well typed by unfolding A into the process associated with A. The types associated with $\overline{u}$ must match those of the global assignment $A : [\overline{S}; n]$ and the rank of the process must be no greater than that of the invocation. The potential mismatch between the two ranks improves typeability in some corner cases. The rules [T-WAIT] and [T-CLOSE] concern processes that exchange termination signals. The channel being closed is consumed and, in the case of [T-WAIT], no longer available in the continuation P. Again, $\mathsf{close}\, u$ can be typed with any rank whereas the rank of $\mathsf{wait}\, u.P$ coincides with that of P. The rules [T-CHANNEL-IN] and [T-CHANNEL-OUT] deal with the exchange of channels in a quite standard way. Note that the actual type of the exchanged channel is required to coincide with the expected one. In particular, no covariance or contravariance of input and output respectively is allowed. Relaxing the typing rule in this way would introduce implicit applications of subtyping that may compromise fair termination [14]. In our type system, each application of subtyping must be explicitly accounted for as we will see when discussing [T-CAST]. Rule

[T-TAG] deals with the exchange of tags. Channels that are not used for such communication must be used in the same way in all branches, whereas the type of the channel on which the message is exchanged changes accordingly. All branches are required to have the same rank, which also corresponds to the rank of the process. Unlike other presentations of this typing rule [23], we require the branches in the process to be matched exactly by those in the type. Again, this is to avoid implicit application of subtyping, which might jeopardize fair termination. The rule [T-CHOICE] deals with non-deterministic choices and requires both continuations to be well typed in the same typing context. The judgment in the conclusion inherits the rank of one of the processes, typically the one with minimum rank. As we will see in Example 6.3, this makes it possible to model finite-rank processes that may create an unbounded number of sessions or that perform an unbounded number of casts.

The rule [T-CAST] models the substitution principle induced by fair subtyping: when $S \leqslant_m T$, a channel of type S can be used where a channel of type T is expected or, in dual fashion [22], a process using u according to T can be used in place of a process using u according to S. To keep track of this cast, the rank in the conclusion is augmented by the weight m of the subtyping relation between S and T. Note that the typing rule guesses the target type of the cast.

Finally, the rule [T-PAR] deals with session creation and parallel composition. This rule is inspired to the *multiparty cut* rule found in linear logic interpretations of multiparty session types [8, 9] and provides a straightforward way for enforcing deadlock freedom. Each process in the composition must be well typed in a slice of the typing context augmented with the endpoint corresponding to its role. The session map of the new session must be coherent, implying that it fairly terminates. The rank of the composition is one plus the aggregated rank of the composed processes, to account for the fact that one more session has been created. Recall that coherence is a property expressed on the LTS of session maps (Definition 4.2) in line with the approach of Scalas and Yoshida [46].

The typing rules described so far are interpreted *coinductively*. That is, in order for a rank n process P to be well typed in Γ there must be a *possibly infinite* derivation tree built with these rules and whose conclusion is the judgment $\Gamma \vdash^n P$. But in a generalized inference system like the one we are defining, this is not enough to establish that P is well typed. In addition, it must be possible to find *finite* derivation trees for all of the judgments occurring in this possibly infinite derivation tree using the discussed rules *and possibly* the corules, which we are about to describe. Since the additional derivation trees must be finite, all of their branches must end up with an application of [T-DONE] or [T-CLOSE], which are the only axioms in Table 6 corresponding to the only terminated processes in Table 1. So, the purpose of these finite typing derivations is to make sure that in every well-typed (sub-)process there exists a path that leads to termination. On the one hand, this is a sensible condition to require as our type system is meant to enforce fair process termination. On the other hand, insisting that these finite derivations can be built using only the typing rules discusses thus far is overly restrictive, for a process might have *one* path that leads to termination, but also alternative paths that lead to (recursive) process invocations. In fact, all of the processes we have discussed in Examples 3.1–3.3 are structured like this. The two corules [CO-CHOICE] and [CO-TAG] in Table 6 establish that, whenever a multi-branch process is dealt with, it suffices for *one* of the branches to lead to termination. A key detail to note in the case of [CO-CHOICE] is that the rank of the non-deterministic choice coincides with that of the branch that leads to termination. This makes sense recalling that the rank associated with a process represents the overall effort required for that process to terminate.

Let us recap the notion of well-typed process resulting from the typing rules of Table 6.

▶ **Definition 5.1** (well-typed process). *We say that P is* well typed *in the context Γ and has rank n if (1) there exists an arbitrary (possibly infinite) derivation tree obtained using the (singly-lined) rules in Table 6 and whose conclusion is $\Gamma \vdash^n P$ and (2) for each judgment in such tree there is a finite derivation obtained using the rules and the (doubly-lined) corules.*

▶ Remark 5.2. The term "*co*rule" seems to suggest that the rule should be *co*inductively interpreted. As we have seen above (Definition 5.1), corules are interpreted inductively. We have chosen to stick with the terminology used in the works that introduced generalized inference systems [3, 17]. ⌟

▶ **Example 5.3.** Let us show some typing derivations for fragments of Example 3.1 using the types S_b, S_s and S_c from Example 4.3. Concerning *Buyer*, we obtain the infinite derivation

$$\dfrac{\dfrac{\dfrac{\vdots}{x : S_b \vdash^0 Buyer\langle x\rangle}\,[\text{T-CALL}]}{x : \mathsf{seller}!\mathsf{add}.S_b \vdash^0 x[\mathsf{seller}]!\mathsf{add}.Buyer\langle x\rangle}\,[\text{T-TAG}] \qquad \dfrac{}{x : !\mathsf{end} \vdash^0 \mathsf{close}\, x}\,[\text{T-CLOSE}]}{x : S_b \vdash^0 x[\mathsf{seller}]!\{\mathsf{add}.x[\mathsf{seller}]!\mathsf{add}.Buyer\langle x\rangle, \mathsf{pay}.\mathsf{close}\, x\}}\,[\text{T-TAG}]$$

and, for each judgment in it, it is easy to find a finite derivation possibly using [CO-TAG]. Concerning *Main* we obtain

$$\dfrac{\dfrac{\vdots}{s[\mathsf{buyer}] : S_b \vdash^0 Buyer\langle s[\mathsf{buyer}]\rangle}\,[\text{T-CALL}] \qquad \dfrac{\vdots}{s[\mathsf{seller}] : S_s \vdash^0 Seller\langle s[\mathsf{seller}]\rangle}\,[\text{T-CALL}] \quad \vdots}{\emptyset \vdash^1 (s)(Buyer\langle s[\mathsf{buyer}]\rangle \mid Seller\langle s[\mathsf{seller}]\rangle \mid Carrier\langle s[\mathsf{carrier}]\rangle)}\,[\text{T-PAR}]$$

where the application of [T-PAR] is justified by the fact that $\mathsf{buyer} \triangleright S_b \mid \mathsf{seller} \triangleright S_s \mid \mathsf{carrier} \triangleright S_c$ is coherent (Example 4.3). No participant creates new sessions or performs casts, so they all have zero rank. The rank of *Main* is 1 since it creates the session s. ⌟

We can prove a strong soundness result for our type system, stating that well-typed, closed processes can always successfully terminate no matter how they reduce.

▶ **Theorem 5.4** (soundness). *If $\emptyset \vdash^n P$ and $P \Rightarrow Q$, then $Q \Rightarrow\preccurlyeq \mathsf{done}$.*

There are several valuable implications of Theorem 5.4 on a well-typed, closed process P:

Deadlock freedom. If Q cannot reduce any further, then it must be (structurally precongruent to) done, namely there are no residual input/output actions.

Fair termination. Under the fairness assumption, Theorem 2.5 assures that P eventually reduces to done. This also implies that every session created by P eventually terminates.

Progress. If Q contains a sub-process with pending input/output actions, the fact that Q may reduce to done means that these actions are eventually performed.

The proof of Theorem 5.4 is essentially composed of a standard subject reduction result showing that typing is preserved by reductions and a proof that every well-typed process other than done may always reduce in such a way that a suitably defined *well-founded measure* strictly decreases. The measure is a lexicographically ordered pair of natural numbers with the following meaning: the first component measures the number of sessions that must be created and the total weight of casts that must be performed in order for the process to terminate (this information is essentially the rank we associate with typing judgments); the second component measures the overall effort required to terminate every session that has already been created (these sessions are identified by the fact that their restriction occurs

unguarded in the process). We account for this effort by measuring the shortest reduction that terminates a coherent session map (Definition 4.2). The reason why we need two quantities in the measure is that in general every application of fair subtyping may *increase* the length of the shortest reduction that terminates a coherent session map. So, when casts are performed the second component of the measure may increase, but the first component reduces. As a final remark, it should be noted that the overall measure associated with a well-typed process *may also increase*, for example if new sessions are created (Example 3.3). However, one particular reduction that decreases the measure is always guaranteed to exist.

We conclude this section discussing a few more examples that motivate the features of the type system that are key for ensuring fair program termination.

▶ **Example 5.5.** To see simple examples of processes whose ill/well typing crucially depends on the fact that we use a generalized inference system consider the definitions

$$A \triangleq A \qquad B \triangleq B \oplus B \qquad C \triangleq C \oplus \mathsf{done}$$

which define a stuck process A, a diverging process B and a fairly terminating process C that admits an infinite reduction. For them we can find the infinite typing derivations below:

$$\dfrac{\dfrac{\vdots}{\emptyset \vdash^0 A}\,[\text{T-CALL}]}{\emptyset \vdash^0 A}\,[\text{T-CALL}] \qquad \dfrac{\dfrac{\vdots \quad \dfrac{\vdots}{\emptyset \vdash^0 B}\,[\text{T-CALL}]}{\emptyset \vdash^0 B \oplus B}\,[\text{T-CHOICE}]}{\emptyset \vdash^0 B}\,[\text{T-CALL}] \qquad \dfrac{\dfrac{\dfrac{\vdots}{\emptyset \vdash^0 C}\,[\text{T-CALL}] \quad \dfrac{}{\emptyset \vdash^0 \mathsf{done}}\,[\text{T-DONE}]}{\emptyset \vdash^0 C \oplus \mathsf{done}}\,[\text{T-CHOICE}]}{\emptyset \vdash^0 C}$$

However, only for C it is possible to find a finite typing derivation using the corule [CO-CHOICE]. So, A and B are ill typed, whereas C is well typed. This is consistent with the fact that only C can always reduce to the successfully terminated process done. ⌟

▶ **Example 5.6** (infinitely ranked processes)**.** The mere existence of a path that leads to termination ensured by the generalized interpretation of the typing rules in Table 6 does not always guarantee that the process is actually able to terminate. An example where this is the case is shown by the process A defined as

$$A \triangleq (s)(s[\mathsf{p}][\mathsf{q}]!\{\mathsf{a}.\mathsf{close}\, s[\mathsf{p}], \mathsf{b}.\mathsf{wait}\, s[\mathsf{p}].A\} \mid s[\mathsf{q}][\mathsf{p}]?\{\mathsf{a}.\mathsf{wait}\, s[\mathsf{q}].A, \mathsf{b}.\mathsf{close}\, s[\mathsf{q}]\})$$

which creates a session s and splits as two parallel sub-processes connected by s. Each sub-process has a path that leads to termination but, because of the way they synchronize, when one sub-process terminates the other one restarts A. For A it would be possible to build a finite typing derivation with the help of [CO-TAG], but A is ill typed because it cannot be assigned a finite rank, since it creates a new session at each recursive invocation.

Further examples of infinitely ranked processes, including ones where the rank is affected by the presence of casts, are discussed by Ciccone and Padovani [14] for binary sessions and can be easily reframed in our multiparty setting. ⌟

6 Advanced Examples

▶ **Example 6.1.** In this example we show that the process $Buyer_1$ playing the role b_1 in the inner session of Example 3.2 is well typed. For clarity, we recall its definition here:

$$\begin{aligned} Buyer_1(x, y) \triangleq y[\mathsf{b}_2]!\{&\mathsf{split}.y[\mathsf{b}_2]?\{\mathsf{yes}.\lceil x \rceil x[\mathsf{s}]!\mathsf{ok}.x[\mathsf{c}]?\mathsf{box}.\mathsf{wait}\, x.\mathsf{wait}\, y.\mathsf{done},\\ &\qquad \mathsf{no}.Buyer_1\langle x, y\rangle\},\\ &\mathsf{giveup}.\mathsf{wait}\, y.\lceil x \rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}\} \end{aligned}$$

We wish to build a typing derivation showing that Buyer_1 has rank 1 and uses x and y respectively according to S and T, where $S = \mathsf{s}!\mathsf{ok}.\mathsf{c}?\mathsf{box}.?\mathsf{end} + \mathsf{s}!\mathsf{cancel}.?\mathsf{end}$ and $T = \mathsf{b}_2!\mathsf{split}.(\mathsf{b}_2?\mathsf{yes}.?\mathsf{end} + \mathsf{b}_2?\mathsf{no}.T) + \mathsf{b}_2!\mathsf{giveup}.?\mathsf{end}$. As it has been noted previously, what makes this process interesting is that it uses the endpoint x differently depending on the messages it exchanges with b_2 on y. Since rule [T-TAG] requires any endpoint other than the one on which messages are exchanged to have the same type, the only way Buyer_2 can be declared well typed is by means of the casts that occur in its body. For the branch in which Buyer_1 proposes to split the payment we obtain the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash^0 \mathsf{done}}\,[\text{T-DONE}]}{y : ?\mathsf{end} \vdash^0 \mathsf{wait}\, y.\mathsf{done}}\,[\text{T-WAIT}]}{x : ?\mathsf{end}, y : ?\mathsf{end} \vdash^0 \mathsf{wait}\, x \dots}\,[\text{T-WAIT}]}{x : \mathsf{c}?\mathsf{box}.?\mathsf{end}, y : ?\mathsf{end} \vdash^0 x[\mathsf{c}]?\mathsf{box}\dots}\,[\text{T-TAG}]}{x : \mathsf{s}!\mathsf{ok}.\mathsf{c}?\mathsf{box}.?\mathsf{end}, y : ?\mathsf{end} \vdash^0 x[\mathsf{s}]!\mathsf{ok}\dots}\,[\text{T-TAG}]}{x : S, y : ?\mathsf{end} \vdash^1 \lceil x \rceil \cdots}\,[\text{T-CAST}] \qquad \dfrac{\vdots}{x : S, y : T \vdash^1 \mathit{Buyer}_1\langle x, y\rangle}\,[\text{T-CALL}]}{x : S, y : \mathsf{b}_2?\mathsf{yes}.?\mathsf{end} + \mathsf{b}_2?\mathsf{no}.T \vdash^1 y[\mathsf{b}_2]?\{\mathsf{yes}\dots, \mathsf{no}\dots\}}\,[\text{T-TAG}]$$

Note how the application of [T-CAST] is key to change the type of x in the branch where the proposed split is accepted by b_2. In that branch, x is deterministically used to send an ok message and we leverage on the fair subtyping relation $S \leqslant_1 \mathsf{s}!\mathsf{ok}.\mathsf{c}?\mathsf{box}.?\mathsf{end}$.

For the branch in which Buyer_1 sends giveup we obtain the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash^0 \mathsf{done}}\,[\text{T-DONE}]}{x : ?\mathsf{end} \vdash^0 \mathsf{wait}\, x.\mathsf{done}}\,[\text{T-WAIT}]}{x : \mathsf{s}!\mathsf{cancel}.?\mathsf{end} \vdash^0 x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}}}{x : S \vdash^1 \lceil x \rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}}\,[\text{T-CAST}]}{x : S, y : ?\mathsf{end} \vdash^1 \mathsf{wait}\, y.\lceil x \rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}}\,[\text{T-WAIT}]$$

Once again the cast is necessary to change the type of x, but this time leveraging on the fair subtyping relation $S \leqslant_1 \mathsf{s}!\mathsf{cancel}.?\mathsf{end}$. These two derivations can then be combined to complete the proof that the body of Buyer_1 is well typed:

$$\dfrac{\vdots \qquad\qquad \vdots}{x : S, y : T \vdash^1 y[\mathsf{b}_2]!\{\mathsf{split}\dots, \mathsf{giveup}\dots\}}\,[\text{T-TAG}]$$

Clearly, it is also necessary to find finite derivation trees for all of the judgments shown above. This can be easily achieved using the corule [CO-TAG]. ⌟

▶ **Example 6.2.** Casts can be useful to reconcile the types of a channel that is used differently in different branches of a non-deterministic choice. For example, below is an alternative modeling of *Buyer* from Example 3.1 where we abbreviate seller to s for convenience:

$$B(x) \triangleq \lceil x \rceil x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle \oplus \lceil x \rceil x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x$$

Note that x is used for sending two add messages in the left branch of the non-deterministic choice and for sending a single pay message in the right branch. Given the session type $S = \mathsf{s}!\mathsf{add}.S + \mathsf{s}!\mathsf{pay}.!\mathsf{end}$ and using the fair subtyping relations $S \leqslant_2 \mathsf{s}!\mathsf{add}.\mathsf{s}!\mathsf{add}.S$ and $S \leqslant_1 \mathsf{s}!\mathsf{pay}.!\mathsf{end}$ we can obtain the following typing derivation for the body of B:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x : S \vdash^1 B\langle x\rangle}\,[\text{T-CALL}]}{x : \mathsf{s}!\mathsf{add}.S \vdash^1 x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle}\,[\text{T-TAG}]}{x : \mathsf{s}!\mathsf{add}.\mathsf{s}!\mathsf{add}.S \vdash^1 x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle}\,[\text{T-TAG}]}{x : S \vdash^3 \lceil x\rceil x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle}\,[\text{T-CAST}] \quad \dfrac{\dfrac{\dfrac{}{x : !\mathsf{end} \vdash^0 \mathsf{close}\, x}\,[\text{T-CLOSE}]}{x : \mathsf{s}!\mathsf{pay}.!\mathsf{end} \vdash^0 x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x}\,[\text{T-TAG}]}{x : S \vdash^1 \lceil x\rceil x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x}\,[\text{T-CAST}]$$
$$\overline{x : S \vdash^1 \lceil x\rceil x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle \oplus \lceil x\rceil x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x}\,[\text{T-CHOICE}]$$

In general, the transformation $u[\mathsf{p}]!\{\mathsf{m}_i.P_i\}_{i=1..n} \rightsquigarrow \lceil u\rceil u[\mathsf{p}]!\mathsf{m}_1.P_1 \oplus \cdots \oplus \lceil u\rceil u[\mathsf{p}]!\mathsf{m}_n.P_n$ does not always preserve typing, so it is not always possible to encode the output of tags using casts and non-deterministic choices. As an example, the definition

$$\mathit{Slot}(x) \stackrel{\triangle}{=} x[\mathsf{player}]?\{\mathsf{play}.x[\mathsf{player}]!\{\mathsf{win}.\mathit{Slot}\langle x\rangle, \mathsf{lose}.\mathit{Slot}\langle x\rangle\}, \mathsf{quit}.\mathsf{close}\, x\}$$

implements the unbiased slot machine of Example 4.6. It is easy to see that *Slot* is well typed under the global type assignment $\mathit{Slot} : [T; 0]$ where $T = \mathsf{player}?\mathsf{play}.(\mathsf{player}!\mathsf{win}.T + \mathsf{player}!\mathsf{lose}.T) + \mathsf{player}?\mathsf{quit}.!\mathsf{end}$. In particular, *Slot* has rank 0 since it performs no casts and it creates no sessions. If we encode the tag output in *Slot* using casts and non-deterministic choices we end up with the following process definition, which is ill typed because it cannot be given a finite rank:

$$\mathit{Slot}(x) \stackrel{\triangle}{=} x[\mathsf{player}]?\{\mathsf{play}.(\lceil x\rceil x[\mathsf{player}]!\mathsf{win}.\mathit{Slot}\langle x\rangle \oplus \lceil x\rceil x[\mathsf{player}]!\mathsf{lose}.\mathit{Slot}\langle x\rangle), \mathsf{quit}.\mathsf{close}\, x\}$$

The difference between this version of *Slot* and the above definition of B is that *Slot* always recurs after a cast, so it is not obvious that finitely many casts suffice in order for *Slot* to terminate. ⌟

▶ **Example 6.3.** Here we provide evidence that the process definitions in Example 3.3 are well typed, even if they model processes that can open arbitrarily many sessions. In that example, the most interesting process definition is that of the worker *Sort*, which is recursive and may create a new session. In contrast, *Merge* is finite and *Main* only refers to *Sort*. We claim that these process definitions are well typed under the global type assignments

$$\mathit{Main} : [(); 1] \qquad \mathit{Sort} : [U; 0] \qquad \mathit{Merge} : [T, V; 0]$$

where $T = \mathsf{m}!\mathsf{res}.!\mathsf{end}$, $U = \mathsf{m}?\mathsf{req}.T$ and $V = \mathsf{w}_1!\mathsf{req}.\mathsf{w}_2!\mathsf{req}.\mathsf{w}_1?\mathsf{res}.\mathsf{w}_2?\mathsf{res}.?\mathsf{end}$.

For the branch of *Sort* that creates a new session we obtain the derivation tree

$$\dfrac{\dfrac{\vdots}{x : T, t[\mathsf{m}] : V \vdash^0 \mathit{Merge}\langle x, t[\mathsf{m}]\rangle}\,[\text{T-CALL}] \quad \dfrac{\vdots}{t[\mathsf{w}_i] : U \vdash^0 \mathit{Sort}\langle t[\mathsf{w}_i]\rangle}\,[\text{T-CALL}], i = 1, 2}{x : T \vdash^1 (t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_2]\rangle)}\,[\text{T-PAR}]$$

where the rank 1 derives from the fact that the created session involves three zero-ranked participants. For the body of *Sort* we obtain the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{x : T \vdash^1 (t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \cdots)}\,[\text{T-PAR}] \quad \dfrac{\dfrac{}{x : !\mathsf{end} \vdash^0 \mathsf{close}\, x}\,[\text{T-CLOSE}]}{x : T \vdash^0 x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\, x}\,[\text{T-TAG}]}{x : T \vdash^0 (t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \cdots) \oplus x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\, x}\,[\text{T-CHOICE}]}{x : U \vdash^0 x[\mathsf{m}]?\mathsf{req}.((t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \cdots) \oplus x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\, x)}\,[\text{T-TAG}]$$

In the application of the rule [T-CHOICE], the rank of the whole choice coincides with that of the branch in which no new sessions are created. This way we account for the fact that, even though *Sort* *may* create a new session, it does not *have to* do so in order to terminate. ⌟

7 Related Work

Fair termination of binary sessions. Our type system is both a refinement and an extension of the one presented by Ciccone and Padovani [14], which ensures the fair termination of *binary* sessions. The main elements of the two type systems are closely related, but there are some key differences. In that work, the fairness assumption being made is *strong fairness* [21, 4, 36, 47] which guarantees fair termination of binary sessions *at the level of types* but not necessarily *at the level of processes*. The key difference between types and processes is that types generate *finite-state* reduction systems (because of their regularity) whereas processes may generate *infinite-state* reduction systems. While strong fairness is known to be the strongest possible fairness assumption for finite-state systems [48], it is not strong enough to make the right-to-left direction of Theorem 2.5 hold for infinite-state systems. In fact, it can be shown that strong fairness and the fairness assumption we make in this work (Definition 2.2) are unrelated for infinite-state reduction systems, in the sense that there exist fair runs that are not strongly fair and there exist strongly fair runs that are not fair runs. The fairness assumption we make in this work is general enough so that it can be related to both types (Definition 4.2) and processes (Theorem 5.4) through Theorem 2.5. The main advantage of working with native multiparty sessions is that they enable the natural modeling of interactions involving multiple participants in possibly cyclic network topologies, like those in Examples 3.2 and 3.3. Another difference and contribution of our work compared to the one of Ciccone and Padovani [14] is that the definition of the fair subtyping relation is simpler. In particular, the inference system we provide (Table 5) does not make use of corules [13, 14] nor does it require auxiliary predicates [40, 42].

Liveness properties of multiparty sessions. The enforcement of liveness properties has always been a key aspect of session type systems, although previous works have almost exclusively focused on progress rather than on (fair) termination. Scalas and Yoshida [46] define a general framework for ensuring safety and liveness properties of multiparty sessions. In particular, they define a hierarchy of three liveness predicates to characterize "live" sessions that enjoy progress. They also point out that the coarsest liveness property in this hierarchy, which is the one more closely related to fair termination, cannot be enforced by their type system. In part, this is due to the fact that their type system relies on a standard subtyping relation for session types [23] instead of fair subtyping [40, 42]. As we have seen in Section 5, even for single-session programs the mere adoption of fair subtyping is not enough and it is necessary to meet additional requirements (Examples 5.5 and 5.6). The work of van Glabbeek et al. [48] presents a type system for multiparty sessions that ensures progress and is not only sound but also complete. The fairness assumption they make – called *justness* – is substantially weaker than our own (Definition 2.2) and such that the unfair runs are those in which some interactions between participants are systematically discriminated in favor of other interactions involving a disjoint set of independent participants. For this reason, their progress property is in between the two more restrictive liveness predicates of Scalas and Yoshida [46] and can only be guaranteed when it is independent of the behavior of the other participants of the same session. In the end, simple sessions like those described in Examples 3.1–3.3 fall outside the scope of these works as far as liveness properties are concerned.

Another major difference between our work and the ones cited above [46, 48] is that fair termination, unlike progress, enables compositional reasoning and so we are able to enforce a global liveness property (Theorem 5.4) *even in the presence of multiple sessions*

(see Examples 3.2 and 3.3). Notable examples of multiparty session type systems ensuring progress also in the presence of multiple (possibly interleaved) sessions are provided by Padovani et al. [43] and by Coppo et al. [15]. This is achieved by a rich type structure that prevents mutual dependencies between different sessions. In any case, these works do not address sessions in which progress may depend on choices made by session participants.

Termination of binary sessions. Termination is a liveness property that can be guaranteed when finite session types are considered [44]. As soon as infinite session types are considered, many session type systems weaken the guaranteed property to deadlock freedom. Lindley and Morris [38] define a type system for a functional language with session primitives and recursive session types that is strongly normalizing. That is, a well-typed program along with all the sessions it creates is guaranteed to terminate. This strong result is due to the fact that the type language is equipped with least and greatest fixed point operators that are required to match each other by duality. Termination is strictly stronger than fair termination. In particular, there exist fairly terminating programs that are not terminating because they allow reductions of unbounded length (see Examples 3.1–3.3).

Liveness properties in the π-calculus. Kobayashi [29] defines a behavioral type system that guarantees lock freedom in the π-calculus. Lock freedom is a liveness property akin to progress for sessions, except that it applies to *any* communication channel (shared or private). Padovani [41] adapts and extends the type system of Kobayashi [29] to enforce lock freedom in the *linear π-calculus* [32], into which binary sessions can be encoded [19]. All of these works annotate types with numbers representing finite upper bounds to the number of interactions needed to unblock a particular input/output action. For this reason, none of our key examples (Examples 3.1–3.3) is in the scope of these analysis techniques. Kobayashi and Sangiorgi [33] show how to enforce lock freedom by combining deadlock freedom and termination. Our work can be seen as a generalization of this approach whereby we enforce lock freedom by combining deadlock freedom (through a mostly conventional session type system) and *fair* termination. Since fair termination is coarser than termination, the family of programs for which lock freedom can be proved is larger as well.

Deadlock freedom. Our type system enforces deadlock freedom essentially thanks to the shape of the rule [T-PAR] which is inspired to the cut rule of linear logic. This rule has been applied to session type systems for binary sessions [49, 6, 38] and subsequently extended to multiparty sessions [8, 9]. In the latter case, the rule – dubbed *multiparty cut* – requires a coherence condition among cut types establishing that the session types followed by the single participants adhere to a so-called global type describing the multiparty session as a whole. The rule [T-PAR] adopts with schema, except that the coherence condition is stronger to entail fair session termination. The key principle of these formulations of the cut rule as a typing rule for parallel processes is to impose a tree-like network topology, whereby two parallel processes can share at most one channel. In the multiparty case, cyclic network topologies can be modeled within each session (Example 3.3) since coherence implies deadlock freedom.

Having a single construct that merges session restriction and parallel composition allows for a simple formulation of the typing rules so that dealock freedom is easily guaranteed. However, many session calculi separate these two forms in line with the original presentation of the π-calculus. We think that our type system can be easily reformulated to support distinct session restriction and parallel composition by means of hypersequents [34, 35].

A more liberal version of the cut rule, named multi-cut and inspired to Gentzen's "mix" rule, is considered by Abramsky et al. [1] enabling processes to share more than one channel. In this setting, deadlock freedom is lost but can be recovered by means of a richer type structure that keeps track of the dependencies between different channels. This approach has been pioneered by Kobayashi [29, 30] for the π-calculus and later on refined by Padovani [41]. Other approaches to ensure deadlock freedom based on *dependency/connectivity graphs* that capture the network topology implemented by processes have been studied by Carbone and Debois [7], Kobayashi and Laneve [31], de'Liguoro and Padovani [20], and Jacobs et al. [28].

8 Concluding Remarks

Sessions ought to terminate. Until recently this property has been granted only for sessions whose duration is bounded. In this work we have presented the first type system ensuring the *fair termination* of multiparty sessions, that is a termination property under the assumption that, if termination is always reachable, then it is eventually achieved. Fair termination is stronger than weak termination but substantially weaker than strong normalization. In particular, fair termination does not rule out infinite runs of well-typed processes as long as they purposefully eschew termination. When fair termination is combined with the usual safety properties of sessions, it entails a strong progress property whereby *any* pending action is eventually performed. Our type system is the first ensuring such strong progress property for multiparty (and possibly multiple) sessions.

A cornerstone element of the type system is *fair subtyping*, a coherence-preserving refinement of the standard subtyping relation for session types [23]. In this work, we have also contributed a new characterization of fair subtyping (Table 5 and Theorems 4.8 and 4.10) that is substantially simpler than previous ones [40, 42, 13, 14] since it does not require auxiliary predicates nor the use of a generalized inference system [3, 17, 13, 14]. Thanks to this new characterization we have been able to prove the transitivity of fair subtyping (Theorem 4.7) without relying on its (relative) completeness with respect to its semantic counterpart (Definition 4.4).

The decidability of fair subtyping and of type checking follow from analogous results for binary sessions [14]. The rank of processes can be inferred using the same algorithm that works for the binary case [14, auxiliary material]. Considering that fair subtyping for multiparty session types coincides with fair subtyping for binary session types except for the presence of roles, it would be easy to adapt the type checking tool FairCheck [12] to the process language we consider in this paper. The most relevant difference would be the algorithm for deciding the coherence of a session map, which is somewhat more complex than that for the compatibility between two session types. As for the binary setting, to which extent the type system is amenable to full type reconstruction is yet to be established. In particular, a hypothetical type inference algorithm would have to be able to solve fair subtyping inequations and this problem has not been investigated yet. Another open question that may have a relevant practical impact is whether the type system remains *sound* in a setting where communications are *asynchronous*. We expect the answer to be positive, as is the case for other synchronous multiparty session types systems [46], but we have not worked out the details yet.

In this paper we have focused on the theoretical aspects of fairly terminating multiparty sessions. A natural development of this work is its application to a real programming environment. We envision two approaches that can be followed to this aim. A bottom-up approach may apply our static analysis technique to a program (in our process calculus) that is extracted from actual code and that captures the code's communication semantics. We expect that suitable annotations may be necessary to identify those branching parts

of the code that represent non-deterministic choices in the program. Most typically, these branches will correspond to finite loops or to queries made to the human user of the program that have several different continuations. A top-down approach may provide programmers with a *generative tool* that, starting from a global specification in the form of a *global type* [26], produces template code that is "well-typed by design" and that the programmer subsequently instantiates to a specific application. Scribble [50, 2] is an example of such a tool. Interestingly, the usual notion of global type projectability is not sufficient to entail that the session map resulting from a projection is coherent. However, coherence would be guaranteed by requiring that the projected global type is fairly terminating.

Finally, we plan to investigate the adaptation of the type system for ensuring the fair termination in the popular actor-based model. This is a drastically different setting in which the order of messages is not as controllable as in the case of sessions. As a consequence, type based analyses require radically different formalisms such as mailbox types [20], for which the study of fair subtyping and of type systems enforcing fair termination is unexplored.

References

1. Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. Interaction categories and the foundations of typed concurrent programming. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 35–113, 1996.
2. Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.
3. Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.
4. Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 189–198. ACM Press, 1987. doi:10.1145/41625.41642.
5. Mario Bravetti and Gianluigi Zavattaro. A theory of contracts for strong service compliance. *Math. Struct. Comput. Sci.*, 19(3):601–638, 2009. doi:10.1017/S0960129509007658.
6. Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.
7. Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In Simon Bliudze, Roberto Bruni, Davide Grohmann, and Alexandra Silva, editors, *Proceedings Third Interaction and Concurrency Experience: Guaranteed Interaction, ICE 2010, Amsterdam, The Netherlands, 10th of June 2010*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.
8. Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

9 Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54(3):243–269, 2017. doi:10.1007/s00236-016-0285-y.

10 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair Termination of Multiparty Sessions. Technical report, Università di Torino and Università di Genova, 2022. URL: https://arxiv.org/abs/2205.08786.

11 Luca Ciccone, Francesco Dagnino, and Elena Zucca. Flexible coinduction in agda. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.13.

12 Luca Ciccone and Luca Padovani. FairCheck. GitHub repository, 2021. URL: https://github.com/boystrange/FairCheck.

13 Luca Ciccone and Luca Padovani. Inference Systems with Corules for Fair Subtyping and Liveness Properties of Binary Session Types. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP'21)*, volume 198 of *LIPIcs*, pages 125:1–125:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.125.

14 Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi:10.1145/3498666.

15 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.*, 26(2):238–302, 2016. doi:10.1017/S0960129514000188.

16 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

17 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Log. Methods Comput. Sci.*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.

18 Francesco Dagnino. *Flexible Coinduction*. PhD thesis, DIBRIS, University of Genoa, January 2021.

19 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

20 Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.15.

21 Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986. doi:10.1007/978-1-4612-4886-6.

22 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.

23 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.

24 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

25 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

26 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

27 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

28 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. doi:10.1145/3498662.

29 Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002. doi:10.1006/inco.2002.3171.

30 Naoki Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006. doi:10.1007/11817949_16.

31 Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017. doi:10.1016/j.ic.2016.03.004.

32 Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999. doi:10.1145/330249.330251.

33 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5):16:1–16:49, 2010. doi:10.1145/1745312.1745313.

34 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018*, volume 292 of *EPTCS*, pages 90–103, 2018. doi:10.4204/EPTCS.292.5.

35 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: a fully-abstract semantics for classical processes. *Proc. ACM Program. Lang.*, 3(POPL):24:1–24:29, 2019. doi:10.1145/3290337.

36 M.Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989. doi:10.1016/0950-5849(89)90159-6.

37 Leslie Lamport. Fairness and hyperfairness. *Distributed Comput.*, 13(4):239–245, 2000. doi:10.1007/PL00008921.

38 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 434–447. ACM, 2016. doi:10.1145/2951913.2951921.

39 Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982. doi:10.1145/357172.357178.

40 Luca Padovani. Fair subtyping for open session types. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2013. doi:10.1007/978-3-642-39212-2_34.

41 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

42 Luca Padovani. Fair subtyping for multi-party session types. *Math. Struct. Comput. Sci.*, 26(3):424–464, 2016. doi:10.1017/S096012951400022X.

43 Luca Padovani, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. Typing liveness in multiparty communicating systems. In eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8459 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2014. doi:10.1007/978-3-662-43376-8_10.

44 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 539–558. Springer, 2012. doi:10.1007/978-3-642-28869-2_27.

45 Jean-Pierre Queille and Joseph Sifakis. Fairness and related properties in transition systems - A temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983. doi:10.1007/BF00265555.

46 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

47 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.

48 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make session types complete for lock-freedom. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470531.

49 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

50 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3

Guillermina Cledou
HASLab, INESC TEC, Porto, Portugal
University of Minho, Braga, Portugal

Luc Edixhoven
Open University of the Netherlands, Heerlen, The Netherlands
NWO-I, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

Sung-Shik Jongmans[1]
Open University of the Netherlands, Heerlen, The Netherlands
NWO-I, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

José Proença
CISTER, ISEP, Polytechnic Institute of Porto, Portugal

Abstract

Construction and analysis of distributed systems is difficult. Multiparty session types (MPST) constitute a method to make it easier. The idea is to use type checking to statically prove deadlock freedom and protocol compliance of communicating processes. In practice, the premier approach to apply the MPST method in combination with mainstream programming languages has been based on API generation. In this paper (pearl), we revisit and revise this approach.

Regarding our "revisitation", using Scala 3, we present the existing API generation approach, which is based on deterministic finite automata (DFA), in terms of both the existing states-as-classes encoding of DFAs as APIs, and a new states-as-type-parameters encoding; the latter leverages match types in Scala 3. Regarding our "revision", also using Scala 3, we present a new API generation approach that is based on sets of pomsets instead of DFAs; it crucially leverages match types, too. Our fresh perspective allows us to avoid two forms of combinatorial explosion resulting from implementing concurrent subprotocols in the DFA-based approach. We implement our approach in a new API generation tool.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Concurrency, pomsets (partially ordered multisets), match types, Scala 3

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.27

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.19`

Funding *G. Cledou and J. Proença*: European Regional Development Fund (ERDF), Operational Programme for Competitiveness and Internationalisation (COMPETE 2020): POCI-01-0145-FEDER-029946 (DaVinci). *S. Jongmans*: Netherlands Organisation of Scientific Research: 016.Veni.192.103. *J. Proença*: Fundação para a Ciência e a Tecnologia (FCT), within the CISTER Research Unit: UIDP/UIDB/04234/2020. ERDF and FCT, Portugal 2020 Partnership Agreement, Norte Portugal Regional Operational Programme (NORTE 2020): NORTE-01-0145-FEDER-028550 (REASSURE). ECSEL Joint Undertaking (JU): grant agreement No 876852 (VALU3S).

[1] Corresponding author

© Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença;
licensed under Creative Commons License CC-BY 4.0

Figure 1 MPST method.

Figure 2 Seller–buyer protocol (Example 1).

Figure 3 Master–workers protocol (Example 2).

1 Introduction

Background. Construction and analysis of distributed systems is difficult. One of the key challenges is to verify absence of communication errors, by proving *deadlock freedom* (i.e., the processes can always terminate or reduce) and *protocol compliance* (i.e., if the processes can terminate or reduce, then the protocol allows it). *Multiparty session types* (MPST) [18,19] constitute a method to overcome these challenges. The idea is visualised in Figure 1:

1. First, a *protocol* among *roles* $r_1, \ldots, r_n$ is implemented as a *session* of *processes* $P_1, \ldots, P_n$ (concrete), while it is specified as a *global type* G (abstract). The global type models the behaviour of all processes, collectively, from their shared perspective (e.g., "first, a number from Alice to Bob; next, a boolean from Bob to Carol").
2. Next, G is decomposed into local types $L_1, \ldots, L_n$, by *projecting* G onto every role. Every local type models the behaviour of one process, individually, from its own perspective (e.g., for Bob, "first, he receives a number from Alice; next, he sends a boolean to Carol").
3. Last, absence of communication errors is verified, by *type-checking* every process P_i against local type L_i. MPST theory guarantees that well-typedness at compile-time (statically) implies deadlock freedom and protocol compliance at execution-time (dynamically).

The following two examples [6,34] further illustrate global/local types in the MPST method.

▶ **Example 1** (seller–buyer [6]). In the *seller–buyer protocol*, visualised in Figure 2, first, the *seller* (**s**) tells the *buyer* (**b**) the description of an item (`Descr`) and a price (`Price`); next, the buyer tells the seller whether it accepts the offer (`Acc`) or rejects it (`Rej`). The following global type specifies the protocol from the seller's and the buyer's shared perspective:

$$G = \mathbf{s}{\twoheadrightarrow}\mathbf{b}{:}\texttt{Descr} \,.\, \mathbf{s}{\twoheadrightarrow}\mathbf{b}{:}\texttt{Price} \,.\, \mathbf{s}{\twoheadrightarrow}\mathbf{b}{:}\{\texttt{Acc}, \texttt{Rej}\} \,.\, \mathbf{end}$$

In this notation, $p{\twoheadrightarrow}q{:}\{t_i . G_i\}_{1 \le i \le n}$ specifies the communication of a value of type t_i from role p to role q, followed by G_i, for some $1 \le i \le n$. We write $p{\twoheadrightarrow}q{:}\{t_i\}_{1 \le i \le n} . G$ as a macro for $p{\twoheadrightarrow}q{:}\{t_i . G\}_{1 \le i \le n}$, and we omit braces when $n = 1$. The following local types (projected from the global type) specify the protocol from the seller's and the buyer's own perspectives:

$$L_{\mathbf{s}} = \mathbf{sb}!\texttt{Descr} \,.\, \mathbf{sb}!\texttt{Price} \,.\, \mathbf{sb}?\{\texttt{Acc}, \texttt{Rej}\} \,.\, \mathbf{end} \qquad L_{\mathbf{b}} = \ldots$$

In this notation, $pq!\{t_i . L_i\}_{1 \le i \le n}$ and $pq?\{t_i . L_i\}_{1 \le i \le n}$ specify the send and receive of a value of type t_i from role p to role q, followed by L_i, for some $1 \le i \le n$. We write $pq!\{t_i\}_{1 \le i \le n} . L$ and $pq?\{t_i\}_{1 \le i \le n} . L$ as macros for $pq!\{t_i . L\}_{1 \le i \le n}$ and $pq?\{t_i . L\}_{1 \le i \le n}$, and we omit braces when $n = 1$. Henceforth, also, we usually omit "$.\mathbf{end}$". ⌟

Figure 4 Workflow of API generation (the first three arrows are performed automatically by the tool; the last arrow is performed manually by the programmer).

▶ **Example 2** (master–workers [34]). In the *master–workers protocol*, visualised in Figure 3, first, the *master* ($\mathtt{m}$) tells two *workers* ($\mathtt{w}_1, \mathtt{w}_2$) to perform work (`Work`); next, the workers tell the master that they are done (`Done`). The following global/local types specify the protocol:

$$G = \mathtt{m} \twoheadrightarrow \mathtt{w}_1 : \mathtt{Work} \,.\, \mathtt{m} \twoheadrightarrow \mathtt{w}_2 : \mathtt{Work} \,.\, \mathtt{w}_1 \twoheadrightarrow \mathtt{m} : \mathtt{Done} \,.\, \mathtt{w}_2 \twoheadrightarrow \mathtt{m} : \mathtt{Done} \qquad L_{\mathtt{m}} = \mathtt{m}\mathtt{w}_1 ! \mathtt{Work} \,.\, \mathtt{m}\mathtt{w}_2 ! \mathtt{Work} \,.\, \mathtt{w}_1\mathtt{m} ? \mathtt{Done} \,.\, \mathtt{w}_2\mathtt{m} ? \mathtt{Done} \qquad L_{\mathtt{w}_1} = \mathtt{m}\mathtt{w}_1 ? \mathtt{Work} \,.\, \mathtt{w}_1\mathtt{m} ! \mathtt{Done} \qquad L_{\mathtt{w}_2} = \ldots$$ ⌟

In practice, the premier approach to apply the MPST method in combination with mainstream programming languages has been based on *API generation*. The main ideas, originally conceived by Deniélou, Hu, and Yoshida, are based on the following insights: **(a)** local types can be interpreted "operationally" as *deterministic finite automata* (DFA) [11, 12]; **(b)** DFAs can be encoded as *application programming interfaces* (API), such that well-typed usage of the APIs at compile-time implies deadlock freedom and protocol compliance at execution-time (cf. step 3 of the MPST method) [20, 21]. The corresponding workflow is visualised in Figure 4. API generation has been influential: it is used in the majority of tools that support the MPST method, including Scribble [20], its many dialects/extensions [7, 25, 27, 31, 33, 37, 46], νScr [45], and `mpstpp` [23].

Unsolved: concurrent subprotocols in MPST practice. The global/local types in Example 1 and Example 2 specify *sequential* protocols: there is only a single static order in which the roles are allowed to communicate. Intuitively, however, imposing such a single static order is needlessly restrictive: in the seller–buyer protocol, there is no apparent reason why the `Descr`-message and the `Price`-message should be sent by the seller in that order, while in the master–workers protocol, there is no apparent reason why the `Done`-messages should be received by the master in "worker-id-order". Thus, the specified protocols in these examples are not just sequential; they are *oversequentialised*.

In general, oversequentialisation in global/local types should be avoided for two reasons:

- *Some ordering decisions can be made only at implementation-time*, based on implementation details that are unknown at specification-time. In Example 1, it may be known only at implementation-time that the seller first computes the contents of the `Price`-message and next of the `Descr`-message. Thus, to maximise throughput, the seller should be able to send these messages in this alternative static order as well (forbidden in Example 1).
- *Some ordering decisions can be made only at execution-time*, based on execution details that are unknown both at specification-time and at implementation-time. In Example 2, it is known only at execution-time in which order the workers send the `Done`-messages (depending on how much actual time performing the work takes). Thus, to improve throughput, the master should be allowed to receive those messages in any dynamic order (forbidden in Example 2), which may be different in different executions.

To allow ordering decisions to be made at implementation-time and/or execution-time, oversequentialisation at specification-time should be avoided. In recognition of this issue, several papers on MPST theory feature a more relaxed version of global/local types in which

1. **State explosion:** Interpretations of local types as DFAs (i.e., second arrow in Figure 4) may suffer from combinatorial explosion: in the presence of parallel composition, DFAs may consist of *an exponential number of states* (e.g., with n workers, the DFA of local type $L_{\mathtt{m}}$ in Example 4 has $2^n + n$ states). As a result, both the time to generate APIs, and the space to store them, are prohibitively long/large for many local types with $\|$.
2. **Branch explosion:** Usages of APIs in processes (i.e., fourth arrow in Figure 4) may suffer from combinatorial explosion, too: in the presence of parallel composition, processes may consist of *an exponential number of branches* to achieve well-typedness. As a result, APIs are prohibitively cumbersome to use for many local types with $\|$.

Figure 5 Complications of supporting concurrent subprotocols in MPST practice.

concurrent subprotocols can be specified (e.g., [6,10,11,23,28]). The idea is to supplement the basic prefix operators $p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\leq i\leq n}$, $pq!\{t_i.L_i\}_{1\leq i\leq n}$, and $pq?\{t_i.L_i\}_{1\leq i\leq n}$ in global/local type calculi with operators for *parallel composition* to express free interleaving (i.e., "fork" subprotocols) and *sequential composition* (i.e., "join" subprotocols).

▶ **Example 3** (seller–buyer, relaxed). The following global/local types specify a relaxed version of the seller–buyer protocol in Example 1:

$$G = (\mathbf{s} \twoheadrightarrow \mathbf{b}{:}\mathtt{Descr} \parallel \mathbf{s} \twoheadrightarrow \mathbf{b}{:}\mathtt{Price}) \cdot \mathbf{s} \twoheadrightarrow \mathbf{b}{:}\{\mathtt{Acc}, \mathtt{Rej}\}$$
$$L_{\mathbf{s}} = (\mathbf{sb}!\mathtt{Descr} \parallel \mathbf{sb}!\mathtt{Price}) \cdot \mathbf{sb}?\{\mathtt{Acc}, \mathtt{Rej}\} \quad L_{\mathbf{b}} = \ldots$$

In this notation, $G_1 \parallel G_2$ (resp. $L_1 \parallel L_2$) specifies the parallel composition of G_1 and G_2 (resp. L_1 and L_2) that freely interleaves their communications (resp. sends/receives), while $G_1 \cdot G_2$ (resp. $L_1 \cdot L_2$) specifies the sequential composition of G_1 and G_2 (resp. L_1 and L_2). ⌟

▶ **Example 4** (master–workers, relaxed). The following global/local types specify a relaxed version of the master–workers protocol in Example 2:

$$G = \mathbf{m} \twoheadrightarrow \mathbf{w}_1{:}\mathtt{Work} \,.\, \mathbf{m} \twoheadrightarrow \mathbf{w}_2{:}\mathtt{Work} \,.\, (\mathbf{w}_1 \twoheadrightarrow \mathbf{m}{:}\mathtt{Done} \parallel \mathbf{w}_2 \twoheadrightarrow \mathbf{m}{:}\mathtt{Done}) \qquad L_{\mathbf{m}} = \mathbf{mw}_1!\mathtt{Work} \,.\, \mathbf{mw}_2!\mathtt{Work} \,.\, (\mathbf{w}_1\mathbf{m}?\mathtt{Done} \parallel \mathbf{w}_2\mathbf{m}?\mathtt{Done}) \qquad L_{\mathbf{w}_1}, L_{\mathbf{w}_2} = \ldots \text{ (as in Example 2)}$$

We note that the local types of $L_{\mathbf{w}_1}$ and $L_{\mathbf{w}_2}$ are exactly the same as in Example 2. Thus, the relaxation affects only the master. We also note that the protocol can be relaxed even further by allowing the master to send to the workers in any order; we skip it for simplicity of later examples in this paper (in which we revisit the relaxed master–workers protocol). ⌟

However, while the importance of supporting concurrent subprotocols to avoid over-sequentialisation has been duly recognised in MPST theory [6,10,11,23,28], almost none of the API generation tools offer it in MPST practice [7,20,25,27,31,33,37,46]. Figure 5 explains two major complications; they pertain to the second arrow in Figure 4 ("interpret as") and the fourth arrow ("use in"). The only API generation tool that features parallel composition does not at all address these complications [23] (i.e., it suffers from both forms of combinatorial explosion in Figure 5). Thus, while concurrent subprotocols are supported in MPST theory, they are effectively unsupported in API-generation-based MPST practice.

We note that concurrent subprotocols are effectively supported by some tools that are based on *runtime verification*. For instance, the tool by Demangeon et al. [9] uses an optimised DFA representation (in which "inner" DFAs for subprotocols can be nested inside states of an "outer" DFA) to compactly represent parallel composition; process behaviour

is dynamically monitored against optimised DFAs. Alternatively, the tool by Hamers and Jongmans [16] computes traces of DFAs on-the-fly by dynamically interpreting global types, guided by process behaviour at execution-time, so the full state space is never computed.

Contributions. In this paper (pearl), we present a fresh perspective on API generation: we show how to effectively support concurrent subprotocols for the first time in MPST practice. To achieve this, we leverage two recent advances:

- On the theoretical side, we take advantage of Guanciale–Tuosto's pomset framework [14] to interpret local types as *sets of pomsets* (SOPs) instead of as DFAs. The key benefit of SOPs over DFAs is that parallel composition can be represented in linear time and space. In this way, both complications in Figure 5 can be avoided. Our usage of Guanciale–Tuosto's pomset framework in API generation is novel.
- On the practical side, we take advantage of Scala 3's *match types* (i.e., "lightweight form of dependent typing") [1] to encode SOPs into APIs. The key benefit of match types is that they enable type-level programming; our encoding pivotally relies on these advanced static capabilities (e.g., the encoding cannot be ported to Java). Our usage of Scala 3's match types in API generation is novel. (We note that Scalas et al. also use new features in Scala 3 to support the MPST method [40, 41], but not match types.)

In § 2, we summarise a version MPST theory that includes parallel composition and sequential composition. In § 3, we ***revisit*** API generation by presenting the existing DFA-based version in Scala 3. Besides the existing "states-as-classes" encoding of DFAs, we also present a new "states-as-type-parameters" encoding that uses match types. In § 4, we ***revise*** API generation to avoid the complications in Figure 5 by presenting a new SOP-based version that also uses match types. In § 5, we give a brief overview of our tool.

2 MPST Theory in a Nutshell

In this section, we summarise a *minimal, loop-free core* of Deniélou–Yoshida's version of MPST theory, which includes parallel composition and sequential composition [10]. That is, given the aim of this paper, we omit orthogonal and/or more advanced features from this section (e.g., dynamic channel creation, dynamic process creation, delegation). Regarding "loop-free", we note that many practically relevant protocols do not require loops (e.g., the auction protocol in [10], the ATM protocol in [14], and the OAuth protocol in [22]).

Global types. Let $\mathbb{R} = \{\texttt{alice}, \texttt{bob}, \texttt{carol}, \ldots, \texttt{s}, \texttt{c}, \texttt{m}, \texttt{w}, \ldots\}$ denote the set of all *roles*, ranged over by p, q, r. Let $\mathbb{T} = \{\texttt{Unit}, \texttt{Bool}, \texttt{Nat}, \ldots, \texttt{Descr}, \texttt{Price}, \texttt{Acc}, \texttt{Rej}, \ldots\}$ denote the set of all *data types*, ranged over by t. Let $\mathbb{G}$ denote the set of all *global types*, ranged over by G:

$$G ::= \textbf{end} \mid p{\twoheadrightarrow}q{:}\{t_i.G_i\}_{1\leq i\leq n} \mid G_1 \parallel G_2 \mid G_1 \cdot G_2$$

Informally, these forms of global types have the following meaning:

- Global type **end** specifies the **empty protocol**.
- Global type $p{\twoheadrightarrow}q{:}\{t_i.G_i\}_{1\leq i\leq n}$ specifies the **asynchronous communication** of a value of type t_i through the buffered channel from role p to role q (unbounded), followed by G_i, for some $1 \leq i \leq n$. As additional well-formedness requirements, we stipulate: **(1)** $p \neq q$ (i.e., no self-communication); **(2)** $t_i \neq t_j$, for every $1 \leq i < j \leq n$ (i.e., deterministic continuations). Singleton types (e.g., $\texttt{Acc}, \texttt{Rej}$) can serve as *labels* to communicate choices.

$$\mathbf{end} \upharpoonright r = \mathbf{end}$$
$$p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\leq i\leq n} \upharpoonright r = \begin{cases} pq!\{t_i.(G_i \upharpoonright r)\}_{1\leq i\leq n} & \text{if } p = r \neq q \\ pq?\{t_i.(G_i \upharpoonright r)\}_{1\leq i\leq n} & \text{if } p \neq r = q \\ G_1 \upharpoonright r & \text{if } p \neq r \neq q \text{ and } G_1 \upharpoonright r = \cdots = G_n \upharpoonright r \end{cases}$$
$$(G_1 \oplus G_2) \upharpoonright r = (G_1 \upharpoonright r) \oplus (G_2 \upharpoonright r)$$

Figure 6 Projection of global types.

- Global type $G_1 \parallel G_2$ specifies the **parallel composition** of G_1 and G_2 that freely interleaves their communications. As an additional well-formedness requirement [10], we stipulate $\mathsf{comm}(G_1) \cap \mathsf{comm}(G_2) = \emptyset$ (i.e., distinct communications in distinct subprotocols), where $\mathsf{comm} : \mathbb{G} \to 2^{\mathbb{R}\times\mathbb{R}\times\mathbb{T}}$ is a function that maps every global type to the communications that occur in it, represented as triples of the form (p, q, t).
- Global type $G_1 \cdot G_2$ specifies the **sequential composition** of G_1 and G_2.

Local types and projection. Let $\mathbb{L}$ denote the set of all *local types*, ranged over by L:

$$L ::= \mathbf{end} \mid \underbrace{pq!\{t_i.L_i\}_{1\leq i\leq n}}_{\text{send}} \mid \underbrace{pq?\{t_i.L_i\}_{1\leq i\leq n}}_{\text{receive}} \mid L_1 \parallel L_2 \mid L_1 \cdot L_2$$

These forms of local types have a similar meaning as the corresponding forms of global types. Henceforth, let $\dagger \in \{!, ?\}$ and $\oplus \in \{\parallel, \cdot\}$.

Let $G \upharpoonright r$ denote the *projection* of G onto r. Formally, $\upharpoonright$ is the smallest partial function induced by the equations in Figure 6. The projections of **end**, $G_1 \parallel G_2$, and $G_1 \cdot G_2$ are easy. The projection of $p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\leq i\leq n}$ onto r depends on the contribution of r to the communication: if r is sender (resp. receiver), then the projection specifies a send (resp. receive); if r does not contribute to the communication, and if r has a unique continuation, then the projection is that continuation. The latter means that r is insensitive to which type was communicated (which, as a non-contributor to the communication, r does not know). We note that projection is partial: if the projection of a global type onto one of its roles is undefined, then the global type is unsupported. We also note that, for simplicity and because it does not affect this paper, we use the "plain merge" instead of the "full merge" [39].

Processes and typing rules. Let $\mathbb{V}$ denote the set of all *values*, ranged over by v. Let $\mathbb{X}$ denote the set of all *variables*, ranged over by x. Let $\mathbb{E}$ denote the set of all *expressions*, ranged over by e. Let $\mathbb{P}$ denote the set of all *processes*, ranged over by P:

$$P ::= \mathbf{0} \mid pq!e.P \mid pq?\{x_i{:}t_i.P_i\}_{1\leq i\leq n} \mid P_1 \parallel P_2 \mid P_1 \cdot P_2$$

Informally, these forms of processes have the following meaning:

- Process $\mathbf{0}$ implements the **empty role**.
- Process $pq!e.P$ implements the **asynchronous send** of the value of expression e through the buffered channel from role p to role q, followed by P. Asynchronous sends can be combined with conditional choices to implement *internal choices* by a process. For instance, the following process implements the second part of the buyer in Example 1:

  if `goodOffer(descr,price)` (**bs**!`Acc()`.$\mathbf{0}$) (**bs**!`Rej()`.$\mathbf{0}$)

$$\frac{\mathit{\Gamma} \vdash e : t_k \quad \mathit{\Gamma} \vdash P : L_k}{\mathit{\Gamma} \vdash pq!e.P : pq!\{t_i.L_i\}_{1 \le i \le n}} \text{[SEND]} \qquad \frac{\mathit{\Gamma}, x_i : t_i \vdash P_i : L_i \text{ for every } 1 \le i \le n}{\mathit{\Gamma} \vdash pq?\{x_i{:}t_i.P_i\}_{1 \le i \le n} : pq?\{t_i.L_i\}_{1 \le i \le n}} \text{[RECV]}$$

$$\frac{}{\mathit{\Gamma} \vdash \mathbf{0} : \mathbf{end}} \text{[EMPTY]} \qquad \frac{\mathit{\Gamma} \vdash P_1 : L_1 \quad \mathit{\Gamma} \vdash P_2 : L_2}{\mathit{\Gamma} \vdash P_1 \parallel P_2 : L_1 \parallel L_2} \text{[PAR]} \qquad \frac{\mathit{\Gamma} \vdash P_1 : L_1 \quad \mathit{\Gamma} \vdash P_2 : L_2}{\mathit{\Gamma} \vdash P_1 \cdot P_2 : L_1 \cdot L_2} \text{[SEQ]}$$

Figure 7 Well-typedness of processes.

- Process $pq?\{x_i{:}t_i.P_i\}_{1 \le i \le n}$ implements the **asynchronous receive** of a value of type t_i into variable x_i through the buffered channel from role p to role q, followed by P_i (i.e., type switch on the received value), for some $1 \le i \le n$. Asynchronous receives can be used to implement *external choices* by the environment of a process. For instance, the following process implements the second part of the seller in Example 1: $\texttt{bs}?\{\texttt{x}{:}\texttt{Acc}.\mathbf{0}, \texttt{x}{:}\texttt{Rej}.\mathbf{0}\}$. Thus, through an internal choice and a reciprocal external choice, the sender can "select" a value of a particular type to control whereto the receiver "branches off".
- Process $P_1 \parallel P_2$ implements the parallel composition of P_1 and P_2. We note that $P_1 \parallel P_2$ is intended to implement *one* role (i.e., there is no communication between P_1 and P_2); the only purpose of parallel composition is to allow the sends and receives of P_1 and P_2 to be ordered dynamically at execution-time.
- Process $P_1 \cdot P_2$ implements the sequential composition of P_1 and P_2.

Let $\mathit{\Gamma} \vdash e : t$ denote *well-typedness* of expression e by data type t in environment $\mathit{\Gamma}$. Let $\mathit{\Gamma} \vdash P : L$ denote *well-typedness* of process P by local type L in environment $\mathit{\Gamma}$. Formally, $\vdash$ is the smallest relation induced by the rules in Figure 7. Rule [EMPTY] states that the empty role is well-typed by the empty protocol. Rule [SEND] states that a send is well-typed by $pq!\{t_i.L_i\}_{1 \le i \le n}$ if, for some $1 \le k \le n$, the value to send is well-typed by t_k and the continuation is well-typed by L_k. Dually, rule [RECV] states that a receive is well-typed by $pq?\{t_i.L_i\}_{1 \le i \le n}$ if, for every $1 \le i \le n$, the continuation is well-typed by L_i under the additional assumption that the received value is well-typed by t_i. Thus, a well-typed process needs to be able to consume all specified inputs (i.e., *input-enabledness*), but produce only one specified output. Rules [PAR] and [SEQ] state that a parallel and sequential composition are well-typed if their operands are.

▶ **Theorem 5** (Deniélou–Yoshida [10]). *If G is a well-formed global type in which roles $r_1, \ldots, r_n$ occur, and if $\vdash P_i : (G \restriction r_i)$ for every $1 \le i \le n$, then the session of $P_1, \ldots, P_n$ is deadlock-free and protocol-compliant with respect to G.*

3 DFA-based API Generation

In this section, we ***revisit*** API generation by presenting the existing DFA-based version in Scala 3, using concepts and notation of the previous section. First, we show how local types can be interpreted operationally as DFAs (§ 3.1). Next, we show how DFAs can be encoded as APIs using the existing *"states-as-classes" encoding* (§ 3.2). Last, we also show how DFAs can be encoded as APIs using a new *"states-as-type-parameters" encoding* (§ 3.3). The value of this second encoding is twofold: it yields APIs with a smaller memory footprint, and it gently introduces match types to set the stage for the next section.

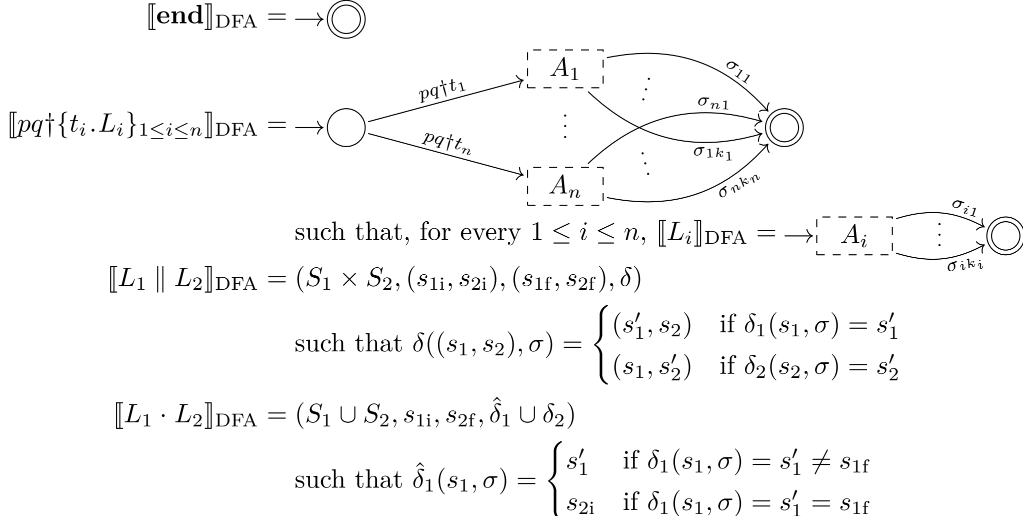


Figure 8 Interpretation of local types as DFAs.

3.1 From Local Types to DFAs

The first interpretation of local types as DFAs was discovered by Deniélou and Yoshida [11]. The key insight is that a local type for a role essentially defines a regular language, each of whose words represents an admissible execution of the role's implementation.

DFAs, formally. Let $\mathbf{\Sigma} = \{pq!t \mid p \neq q\} \cup \{pq?t \mid p \neq q\}$ denote the set of all *type-level actions* ("the alphabet"), ranged over by σ. Let $\mathbb{A}$ denote the set of all *deterministic finite automata* (DFA) over $\mathbf{\Sigma}$, ranged over by A. Formally, a DFA is a tuple $(S, s_\mathrm{i}, s_\mathrm{f}, \delta)$, where S denotes a set of *states*, $s_\mathrm{i}, s_\mathrm{f} \in S$ denote the *initial state* and the *final state*, and $\delta : S \times \mathbf{\Sigma} \rightharpoonup S$ denotes a *transition function*.

Interpretation. Let $[\![L]\!]_\mathrm{DFA}$ denote the *interpretation* of local type L as a DFA. Formally, $[\![\text{-}]\!]_\mathrm{DFA} : \mathbb{L} \to \mathbb{A}$ is the smallest function induced by the equations in Figure 8. The interpretation of **end** is the DFA that accepts the empty language. The interpretation of $pq\dagger\{t_i.L_i\}_{1\leq i\leq n}$ is the DFA that accepts the language of words that begin with $pq\dagger t_i$ and continue with a word accepted by the interpretation of L_i; the visualisation is intended to convey that the final states of the interpretations of $L_1, \ldots, L_n$ are "superimposed" to form a single new final state. The interpretations of $L_1 \parallel L_2$ and $L_1 \cdot L_2$ are the DFAs that accept the shuffle and the concatenation of the languages accepted by the interpretations of L_1 and L_2. We note that the transition function of $[\![L_1 \parallel L_2]\!]_\mathrm{DFA}$ is well-defined due to the well-formedness requirement (§ 2) that the sets of sends and receives that occur in L_1 and L_2 are disjoint (i.e., $\delta_1(s_1, \sigma) = s_1'$ and $\delta_2(s_2, \sigma) = s_2'$ cannot both be true).

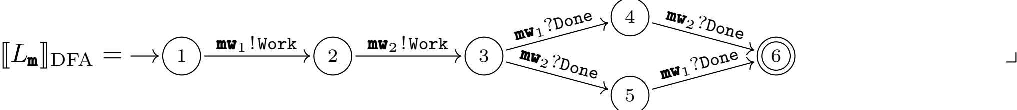
▶ **Example 6.** The following DFA is the interpretation of $L_\mathbf{s}$ in Example 3:

$[\![L_\mathbf{s}]\!]_\mathrm{DFA} =$

1 —**sb**!Descr→ 2 —**sb**!Price→ 4
1 —**sb**!Price→ 3 —**sb**!Descr→ 4
4 —**bs**?Acc→ 5
4 —**bs**?Rej→ 5

◀

▶ **Example 7.** The following DFA is the interpretation of $L_{\mathtt{m}}$ in Example 4:



3.2 From DFAs to APIs – Using Classes

The first encoding of DFAs as APIs was developed by Hu and Yoshida [20]. The key insight is that an input-enabled process P is well-typed by local type L (§ 2) if, and only if, every possible sequence of sends and receives by P forms a word accepted by $[\![L]\!]_{\mathrm{DFA}}$. The "trick", then, is to structure the API in such a way that when the compiler successfully type-checks the API's usage, it has effectively performed an *accepting run* of $[\![L]\!]_{\mathrm{DFA}}$ for every possible sequence of sends and receives by P. In the rest of this subsection, we show how to achieve this in Scala 3 using the existing states-as-classes encoding of DFAs as APIs; experts may skip this subsection, or quickly browse through it only to familiarise themselves with notation.

Suppose that $(S, s_{\mathrm{i}}, s_{\mathrm{f}}, \delta)$ is the interpretation of the local type for role r:

- Every state $s \in S$ is encoded as class $\langle r \rangle$`$`$\langle s \rangle$ in the API, where $\langle r \rangle$ and $\langle s \rangle$ are identifiers for r and s (and `$` is a meaningless separator).
- Every transition $\delta(s, \sigma)$ is encoded as a method of class $\langle r \rangle$`$`$\langle s \rangle$ to perform action σ and provide an instance of class $\langle r \rangle$`$`$\langle \delta(s, \sigma) \rangle$, as detailed shortly.

To use the API, the idea is to define a function `f` that consumes an "initial state object" `s` of type $\langle r \rangle$`$`$\langle s_{\mathrm{i}} \rangle$ as input and produces a "final state object" of type $\langle r \rangle$`$`$\langle s_{\mathrm{f}} \rangle$ as output. Inside of `f`, initially, the only protocol-related actions that can be performed, are those for which `s` has a method. When such a method is called on `s`, an action is performed and a fresh "successor state object" `sNext` is provided. Subsequently, the only protocol-related actions that can be performed, are those for which `sNext` has a method. When such a method is called on `sNext`, another action is performed, and another fresh "successor successor state object" `sNextNext` is provided. This goes on until the final state object is provided.[2] To ensure that every state object is used at most once (see also footnote 2), every state class extends the following trait:

```
trait UseOnce:
  var used = false
  def use = if used then throw new Exception() else used = true
```

When a method is called on a state object `s`, inside of it, method `use` is first called to ensure that `s` has not been used before. Otherwise, an exception is thrown. This technique was first used by Tov and Pucella [42] and has since been adopted in tools for both binary sessions (e.g., [35, 38]) and multiparty sessions (e.g., [7, 20, 23, 31, 33, 37, 45]). We note that `used` in `UseOnce` should actually be declared `private` to shield it from external modification; we omit such access modifiers in our listings for simplicity.

[2] When a method is called on an instance of class $\langle r \rangle$`$`$\langle s \rangle$, *but the method does not exist*, it means that: **(1)** state s in the DFA does not have a corresponding transition; **(2)** hence, the local type for r does not specify the corresponding action; **(3)** hence, the action is not allowed in the protocol. The compiler statically detects this while type-checking and reports an error. As successor state objects become available only after predecessor state objects are used, *and assuming that every state object is used exactly once*, well-typed usage of the API implies protocol compliance. Moreover, as a final state object must have been provided upon termination, *and assuming that there are no other sources of non-terminating or exceptional behaviour*, well-typed usage of the API also implies deadlock freedom. We note that these two additional assumptions cannot be statically enforced using Scala 3's type system (just as with many existing tools [7, 20, 23, 31, 33, 37, 45]): checking the first assumption requires a form of substructural typing, while checking the second assumption is generally undecidable. However, the first assumption can be dynamically monitored using lightweight checks at execution-time.

Next, we explain in more detail how states and transitions can be encoded as classes and methods. As usual in DFA-based API generation, we require that every state in the DFA-interpretation of a local type is either an *output state* with only send transitions, or an *input state* with only receive transitions. In the absence of parallel composition, this requirement is satisfied by construction (§ 3.2), but in the presence of parallel composition, it needs to be checked separately. The classes for output states and input states look as follows:

- Suppose that $(S, s_\mathrm{i}, s_\mathrm{f}, \delta)$ is the interpretation of the local type for role p. Every output state $s \in S$, with transitions $\delta(s, pq_1!t_1), \ldots, \delta(s, pq_n!t_n)$, is encoded as follows:

```
class ⟨p⟩$⟨s⟩(net: Network) extends UseOnce: // output state
  def send(q: ⟨q₁⟩, e: ⟨t₁⟩): ⟨p⟩$⟨δ(s,pq₁!t₁)⟩ = { use; ... /* real send */ }
  ...
  def send(q: ⟨qₙ⟩, e: ⟨tₙ⟩): ⟨p⟩$⟨δ(s,pqₙ!tₙ)⟩ = { use; ... /* real send */ }
```

  Parameter `net` of class $\langle p\rangle$`$`$\langle s\rangle$ encapsulates the underlying communication infrastructure (e.g., shared-memory channels); it is used inside of every `send` method to perform the "real send". Parameter `q` of every `send` method is the identifier of the receiver, parameter `e` is the value to send, and the return value is a fresh successor state object. Roughly, these methods in Scala 3 are typed versions of $pq!e.P$ in the process calculus (§ 2).

- Suppose that $(S, s_\mathrm{i}, s_\mathrm{f}, \delta)$ is the interpretation of the local type for role q. Every input state $s \in S$, with transitions $\delta(s, p_1q?t_1), \ldots, \delta(s, p_nq?t_n)$, is encoded as follows:

```
class ⟨q⟩$⟨s⟩(net: Network) extends UseOnce: // input state
  def recv(f1: (⟨p₁⟩, ⟨t₁⟩, ⟨q⟩$⟨δ(s,p₁q?t₁)⟩) => ⟨q⟩$⟨s_f⟩,
           ...,
           fn: (⟨pₙ⟩, ⟨tₙ⟩, ⟨q⟩$⟨δ(s,pₙq?tₙ)⟩) => ⟨q⟩$⟨s_f⟩) = { use; ... /* real recv */ }
```

  Parameter `f`i of method `recv` is the i-th *continuation*; it is called with the identifier of the sender, the value to receive, and a fresh successor state object after the "real receive". Roughly, this method in Scala 3 is $pq?\{x_i{:}t_i.P_i\}_{1\leq i\leq n}$ in the process calculus (§ 2).

▶ **Example 8.** The following API is the states-as-classes encoding of $[\![L_{\mathtt{s}}]\!]_{\mathrm{DFA}}$ in Example 6:

```
class S$1(net: Network) extends UseOnce:
  def send(q: B, e: Descr): S$2 = ...
  def send(q: B, e: Price): S$3 = ...
class S$2(net: Network) extends UseOnce:
  def send(q: B, e: Price): S$4 = ...
class S$3(net: Network) extends UseOnce:
  def send(q: B, e: Descr): S$4 = ...
```

```
class S$4(net: Network) extends UseOnce:
  def recv(
    f1: (B, Acc, S$5) => S$5,
    f2: (B, Rej, S$5) => S$5): S$5 = ...
class S$5(net: Network) extends UseOnce

type S$Initial = S$1
type S$Final   = S$5
```

The following well-typed function implements the seller:

```
def seller(s: S$Initial): S$Final = s
  .send(B, new Descr).send(B, new Price).recv(
    (q: B, x: Acc, s) => { println("offer accepted"); s },
    (q: B, x: Rej, s) => { println("offer rejected"); s }
```

We note that the two sends in the implementation of the seller can be swapped (i.e., first the price, second the description): the resulting code would still be well-typed, indicating to the programmer that the protocol is not violated. We also note that the type of parameter `s` in the continuations on the last two lines is inferred by the compiler. This demonstrates that the programmer does not need to know how states are represented. The types of parameters `q` and `x` can be inferred as well, so the annotations are redundant; we added them here for clarity of presentation (but will omit them from now on). For details, see: `https://scastie.scala-lang.org/779xP1Z8QwC1DLKDFYAsJg`. ◀

▶ **Example 9.** The following API is the states-as-classes encoding of $[\![L_{\mathtt{m}}]\!]_{\mathrm{DFA}}$ in Example 7:

```
class M$1(net: Network) extends UseOnce:
  def send(q: W1, e: Work): M$2 = ...
class M$2(net: Network) extends UseOnce:
  def send(q: W2, e: Work): M$3 = ...
class M$3(net: Network) extends UseOnce:
  def recv(
    f1: (W1, Done, M$4) => M$6,
    f2: (W2, Done, M$5) => M$6): M$6 = ...
```

```
class M$4(net: Network) extends UseOnce:
  def recv(f2: ... => M$6): M$6 = ...
class M$5(net: Network) extends UseOnce:
  def recv(f1: ... => M$6): M$6 = ...
class M$6(net: Network) extends UseOnce

type M$Initial = M$1
type M$Final = M$6
```

The following well-typed function implements the master:

```
def master(s: M$Initial): M$Final = s
  .send(W1, new Work).send(W2, new Work).recv(
    (_, _, s) => s.recv((_, _, s) => { println("first #1, second #2"); s }),
    (_, _, s) => s.recv((_, _, s) => { println("first #2, second #1"); s }))
```

We note that the two sends in the implementation of the master cannot be swapped (i.e., first to worker 2, second to worker 1): the resulting code would not be well-typed, indicating to the programmer that the protocol is violated. We also note that we omitted all type annotations for parameters in continuations; they can be inferred by the compiler. For details, see: `https://scastie.scala-lang.org/Lg2ZNlw8T7eTfcef7k3yIw`. ⌟

3.3 From DFAs to APIs – Using Type Parameters

In the previous subsection, we showed how the existing *states-as-classes* encoding of DFAs can be used to "trick" the compiler into performing type-level accepting runs. Attractively, states-as-classes requires only basic features of the underlying type system; as a result, it can be applied in combination with a wide range of programming languages (e.g., F# [33], F⋆ [46], Go [7], Java [20], OCaml [45], PureScript [25], Rust [27], Scala [37], TypeScript [31]). In this subsection, we present a new *states-as-type-parameters* encoding of DFAs that leverages an advanced feature of Scala 3's type system: *match types* [1]. While the primary aim of this subsection is to gently explain match types and set the stage for §4, states-as-type-parameters has a technical advantage as well: it is more space-efficient (i.e., states-as-classes requires all specific state classes to be loaded in memory, which can be many, while states-as-type-parameters requires only one generic state class to be loaded.)

Match types. As a brief digression, consider the following example to introduce match types: suppose that we need to write a function that converts `Int`s to `Boolean`s and vice versa. Naively, we could write the signature of this function as follows:

```
type IntOrBoolean = Int | Boolean // type alias for a union type
def convert(x: IntOrBoolean): IntOrBoolean = x match
  case i: Int      => i == 1
  case b: Boolean => if b then 1 else 0
```

The trouble with this first attempt is that the return type, `IntOrBoolean`, is insufficiently precise. For instance, the compiler fails to prove that expression `convert(5) && false` is safe, as it cannot infer that `convert(5)` is of type `Boolean`. Essentially, what the compiler is missing, is a relation between the actual type of `x` (e.g., `Int`) and the return type (e.g., `Boolean`). Match types allow us to define such relations.

1. First, we redefine the signature of `convert` as follows:

   ```
   def convert[T <: IntOrBoolean](x: T): Convert[T] = ... // same as before
   ```

   That is, we introduce a type parameter `T`, which must be a subtype of `IntOrBoolean` (generally, `A` and `B` are subtypes of `A|B`), and we declare `x` to be of type `T`. Furthermore, we declare the return value of the function to be of match type `Convert[T]`. The idea is to define `Convert[T]` in such a way that the intended relation between the actual type of `x` and the return type can be inferred.

2. Next, we define `Convert[T]` as follows:

```
type Convert[T] = T match
  case Int     => Boolean
  case Boolean => Int
```

The compiler *reduces* every occurrence of `Convert[T]` to `Int` or `Boolean`, depending on the instantiation of `T` (e.g., `Convert[Int]` reduces to `Boolean`).

3. Last, for instance, the compiler succeeds/fails to type-check the following expressions:

```
convert(5) + 6          // fail
convert(5) && false     // succeed
```

```
convert(true) + 6       // succeed
convert(true) && false  // fail
```

Thus, match types constitute a "lightweight form of dependent typing" to perform "type-level programming" [1]. In the rest of this subsection, we show how to leverage them in the states-as-type-parameters encoding of DFAs as APIs.

Encoding of DFAs as APIs. The idea behind the states-as-type-parameters encoding is to generate, for every role r, a generic state class $\langle r\rangle$`$State`; it has one type parameter, called `N`. Every instantiation of `N` with a *numeric literal type* (e.g., in Scala, symbol "`5`" denotes both value 5 and a type with value 5 as its only inhabitant) specialises the generic state class into a specific one. For instance, $\langle r\rangle$`$State[1]`, $\langle r\rangle$`$State[2]`, and $\langle r\rangle$`$State[3]` represent three different states, identified by types `1`, `2`, and `3`. Thus, $\langle r\rangle$`$State` has the following header:

```
class ⟨r⟩$State[N](n: N, net: Network) extends UseOnce:
```

This class has two methods: `send` and `recv`. At execution-time, when `send` or `recv` is called, an action is performed and a fresh successor state object is returned. At compile-time, to check that this method call is actually allowed in the current state, the compiler tries to reduce a match type: if it succeeds, the call is allowed; if it fails, it is not.

Regarding `send`, the idea is to use a match type $\langle r\rangle$`$SendReturn` for the *return value*; it has three type parameters, called `N`, `Q`, and `E`, which identify the current state s, the receiver q, and the type t of the value to send. If the DFA has a send transition $\delta(s, rq!t) = s'$, then the compiler succeeds to reduce $\langle r\rangle$`$SendReturn[`$\langle s\rangle$`,`$\langle q\rangle$`,`$\langle t\rangle$`]` to $\langle r\rangle$`$State[`$\langle s'\rangle$`]`; this is the type of the fresh successor state object after sending. In contrast, if the DFA does not have such a transition, then the compiler fails to reduce and yields an error. Thus, `send` looks as follows:

```
  def send[Q, E](q: Q, e: E): ⟨r⟩$SendReturn[N, Q, E] = { use; ... }
```

Regarding `recv`, the idea is to use a match type $\langle r\rangle$`$RecvArgument` for the *argument values*; it has one type parameter, called `N`, which identifies the current state s. If the DFA has receive transitions $\delta(s, p_1 r?t_1) = s'_1, \ldots, \delta(s, p_n r?t_n) = s'_n$, then the compiler succeeds to reduce $\langle r\rangle$`$RecvArgument[`$\langle s\rangle$`]` to a tuple of function types each of which is of the form `(`$\langle p_i\rangle$`,`$\langle t_i\rangle$`,`$\langle r\rangle$`$State[`$\langle s'_i\rangle$`]) =>` $\langle r\rangle$`$Final`; these are the types of the continuations after receiving. In contrast, if the DFA does not have such transitions, then the compiler fails to reduce and yields an error. Thus, `recv` looks as follows:

```
  def recv(fff: ⟨r⟩$RecvArgument[N]): ⟨r⟩$Final = { use; ... }
```

The following examples demonstrate these match types.

▶ **Example 10.** The following API (excerpt) is the states-as-type-parameters encoding of $[\![L_\mathtt{s}]\!]_{\mathrm{DFA}}$ in Example 6 (cf. the states-as-classes encoding in Example 8):

```
type S$SendReturn[N, Q, E] =
  (N, Q, E) match
    case (1, B, Descr) => S$State[2]
    case (1, B, Price) => S$State[3]
    case (2, B, Price) => S$State[4]
    case (3, B, Descr) => S$State[4]
```

```
type S$RecvArgument[N] = N match
  case 4 => (
    (B, Acc, S$State[5]) => S$State[5],
    (B, Rej, S$State[5]) => S$State[5])
```

Every case in match type `S$SendReturn` encodes a send transition out of states 1–3 in the DFA in Example 6. Similarly, the single case in match type `S$RecvArgument` encodes the set of receive transitions out of state 4. We note that exactly the same function that implements the seller in Example 8 is also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/YB9G1KuVQxGkbwgqm7LKLw`. ⌟

▶ **Example 11.** The following API (excerpt) is the states-as-type-parameters encoding of $[\![L_\mathtt{m}]\!]_{\mathrm{DFA}}$ in Example 7 (cf. the states-as-classes encoding in Example 9):

```
type M$SendReturn[N, Q, E] = (N, Q, E) match
  case (1, W1, Work) => M$State[2]
  case (2, W2, Work) => M$State[3]

type M$RecvArgument[N] = N match
  case 3 => ((W1, Done, M$State[4]) => M$State[6],
             (W2, Done, M$State[5]) => M$State[6])
  case 4 => ((W2, Done, M$State[6]) => M$State[6])
  case 5 => ((W1, Done, M$State[6]) => M$State[6])
```

Every case in match type `M$SendReturn` encodes a send transition out of states 1–2 in the DFA for the master in Example 7. Similarly, every case case in match type `M$RecvArgument` encodes a set of receive transitions out of states 3–5. We note that exactly the same functions that implement the master and worker 1 in Example 9 are also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/HHy4TUYLREeQYc8yW6UwHw`. ⌟

We note that states-as-type-parameters fully supports recursive protocols, and it never gives rise to non-terminating compile-time reductions of match types: every reduction only checks if a call to `send` or `recv` on a state object is allowed by considering its finitely many outgoing transitions; possibly infinitely long paths through the DFA are not considered.

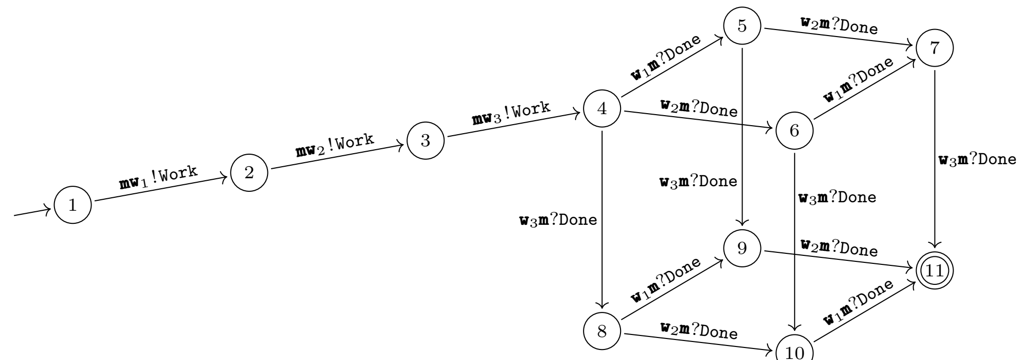
4 SOP-based API Generation

To more clearly demonstrate the complications of supporting concurrent subprotocols with DFA-based API generation (Figure 5), we consider another example.

▶ **Example 12.** The following local type specifies the master in the relaxed master–workers protocol (Example 4), but with three workers instead of two:

$$L_\mathtt{m} = \mathtt{mw}_1\texttt{!Work}\,.\,\mathtt{mw}_2\texttt{!Work}\,.\,\mathtt{mw}_3\texttt{!Work}\,.\,(\mathtt{w}_1\mathtt{m}?\mathrm{Done} \parallel \mathtt{w}_2\mathtt{m}?\mathrm{Done} \parallel \mathtt{w}_3\mathtt{m}?\mathrm{Done})$$

The following DFA is the interpretation of the local type:



This DFA demonstrates complication 1 in Figure 5 (i.e., state explosion): it has $O(2^n)$ states, where $n = 3$ is the number of unordered receives.

The following well-typed function implements the master, using an API that encodes the DFA (whether states-as-classes or states-as-type-parameters is used, is irrelevant here):

```
def master(s: M$Initial): M$Final = s
  .send(W1, new Work).send(W2, new Work).send(W3, new Work).recv(
    (_, _, s) => s.recv(
        (_, _, s) => s.recv(
          (_, _, s) => { println("#1, #2, #3"); s }),
        (_, _, s) => s.recv(
          (_, _, s) => { println("#1, #3, #2"); s })),
    (_, _, s) => s.recv(
        (_, _, s) => s.recv(
          (_, _, s) => { println("#2, #1, #3"); s }),
        (_, _, s) => s.recv(
          (_, _, s) => { println("#2, #3, #1"); s })),
    (_, _, s) => s.recv(
        (_, _, s) => s.recv(
          (_, _, s) => { println("#3, #1, #2"); s }),
        (_, _, s) => s.recv(
          (_, _, s) => { println("#3, #2, #1"); s })))
```

This implementation demonstrates complication 2 in Figure 5 (i.e., branch explosion): it has $O(n!)$ branches (each of which implements a distinct order in which the receives might dynamically take place), where $n = 3$ is the number of unordered receives. ⌟

In this section, we ***revise*** API generation to avoid the complications in Figure 5 by presenting a new version based on *sets of pomsets* (SOP). First, we show how local types can be interpreted operationally as SOPs (§ 4.1). Next, we show how SOPs can be encoded as APIs using match types (§ 4.2).

4.1 From Local Types to SOPs

Our interpretation of local types as SOPs is based on the recent pomset framework by Guanciale and Tuosto [14]. The key insight is that every subset of *words that differ only in the order of concurrent actions* (in the regular language defined by a local type) can be represented as a pomset in exponentially less time and space.

Pomsets, formally (structure). The formalisation of *pomsets* (“partially ordered multisets”) is relatively complicated; we first explain the intuition. Recall that a multiset is a set in which every element can have multiple “instances” (e.g., $\{a, a, b\}$ has two elements but three instances). A pomset is just a multiset endowed with a partial order $\prec$ on the instances (e.g., $a \prec b \prec a$, where the left a and the right a are different instances). Following Pratt and Gischer [13, 36], the idea is to formalise pomsets using labelled digraphs: vertices represent instances, arcs represent the ordering, and vertex labels represent elements (e.g.,

$$\frac{v' \not\prec v \text{ for every } v'}{(V, \prec, \lambda) \xrightarrow{\lambda(v)} (V \backslash \{v\}\,,\, \prec \backslash (\{v\} \times V)\,,\, \lambda \backslash (v \mapsto \lambda(v)))}\,[\textsc{Lpo}] \qquad \frac{X \xrightarrow{\sigma} X'}{[X] \xrightarrow{\sigma} [X']}\,[\textsc{Pom}]$$

$$\frac{\mathcal{X}_i \xrightarrow{\sigma} \mathcal{X}_i' \text{ and } 1 \leq i \leq n}{\{\mathcal{X}_1, \ldots, \mathcal{X}_n\} \xrightarrow{\sigma} \{\mathcal{X}_i'\}}\,[\textsc{SopA}] \quad \text{or} \quad \frac{\mathcal{X}_i \xrightarrow{\sigma} \mathcal{X}_i' \text{ for every } 1 \leq i \leq k \quad \mathcal{X}_i \not\xrightarrow{\sigma} \text{ for every } k+1 \leq i \leq n}{\{\mathcal{X}_1, \ldots, \mathcal{X}_n\} \xrightarrow{\sigma} \{\mathcal{X}_1', \ldots, \mathcal{X}_k'\}}\,[\textsc{SopB}]$$

Figure 9 Transitions of lposets, pomsets, and sets of pomsets.

$\{a, a, b\}$, endowed with $a \prec b \prec a$, can be formalised using labelled vertices $1_a, 2_a, 3_b$ and arcs $(1,3), (3,2)$); such labelled digraphs are called *lposets* ("labelled partially ordered sets"). To make the formalisation of pomsets insensitive to the choice of vertices (e.g., it should not matter if we use $1_a, 2_a, 3_b$ or $4_a, 5_a, 6_b$ as labelled vertices), pomsets are ultimately defined as isomorphism classes of lposets. Henceforth, we write "events" instead of "vertices".

Let **Lpo** denote the set of all *lposets*, ranged over by X. Formally, an lposet is a tuple $(V, \prec, \lambda)$, where V is a set of *events*, $\prec \subseteq V \times V$ is a *precedence relation* (strict partial order), and $\lambda : V \to \mathbf{\Sigma}$ is a *labelling function*, where $\mathbf{\Sigma} = \{pq!t \mid p \neq q\} \cup \{pq?t \mid p \neq q\}$ (§ 3.1). Let $\mathsf{pred}_X(v)$ and $\mathsf{succ}_X(v)$ denote the *immediate predecessors* and *successors* of v in X. Formally, pred and succ are the smallest functions induced by the following equations:

$$\mathsf{pred}_{(V,\prec,\lambda)}(v) = \{v' \mid v \prec v' \text{ and } (v', v) \notin \{(v_1, v_3) \mid v_1 \prec v_2 \prec v_3\}\}$$
$$\mathsf{succ}_{(V,\prec,\lambda)}(v) = \{v' \mid v' \prec v \text{ and } (v, v') \notin \{(v_1, v_3) \mid v_1 \prec v_2 \prec v_3\}\}$$

Let $X_1 \parallel X_2$ and $X_1 \cdot X_2$ denote the *parallel composition* and the *sequential composition* of X_1 and X_2. Formally, $\parallel$ and $\cdot$ are the smallest functions induced by the following equations:

$$(V_1, \prec_1, \lambda_1) \parallel (V_2, \prec_2, \lambda_2) = (V_1 \cup V_2, \prec_1 \cup \prec_2, \lambda_1 \cup \lambda_2)$$
$$(V_1, \prec_1, \lambda_1) \cdot (V_2, \prec_2, \lambda_2) = (V_1 \cup V_2, \prec_1 \cup \prec_2 \cup (V_1 \times V_2), \lambda_1 \cup \lambda_2)$$

Let $X_1 \cong X_2$ denote *isomorphism equivalence* of X_1 and X_2; informally, X_1 and X_2 are isomorphic if, and only if, there exists a bijection between their sets of events that preserves their precedence relations and labelling functions.

Let $\mathbf{Pom} = \mathbf{Lpo}/\cong$ denote the set of all *pomsets*, ranged over by $\mathcal{X}$; it is the quotient set of **Lpo** by $\cong$ (i.e., a pomset is an isomorphism class of lposets). We write $[X]$ to denote the isomorphism class of X (i.e., a pomset). A set of pomsets is similar to a formal language (i.e., a set of words), except that words that differ only in the order of concurrent actions are represented collectively as a single pomset instead of individually as multiple words.

Pomsets, formally (behaviour). A transition relation can be associated to lposets, pomsets, and sets of pomsets. Figure 9 shows the rules. Rule [Lpo] states that an lposet can reduce with symbol $\lambda(v)$ when v is minimal; v is removed from the reduced lposet. Rule [Pom] states that a pomset can reduce with symbol σ when one of its lposets can.

Rule [SopA] states that a set of pomsets can reduce when one of its pomsets can; the reduced pomset is kept, while the others are removed. This formalises the idea of an *early choice*: at the start of an execution, when the first action is performed, a commitment is made to one behaviour. Alternatively, rule [SopB] states that a set of pomsets can reduce with symbol σ when it can be split into two disjoint subsets, such that k pomsets can reduce with σ, while the remaining $n - k$ pomsets cannot; the k reduced pomsets are kept, while the others are removed. This formalises the idea of a *late choice*: during an execution, as

$$
\begin{aligned}
[\![\mathbf{end}]\!]_{\text{SOP}} &= \{[(\emptyset,\emptyset,\emptyset)]\} \\
[\![pq\dagger\{t_i.L_i\}_{1\le i\le n}]\!]_{\text{SOP}} &= \left\{\left[\overset{pq\dagger t_i}{\bullet}\longrightarrow \boxed{X}\right] \,\middle|\, [X]\in[\![L_i]\!]_{\text{SOP}} \text{ and } 1\le i\le n\right\} \\
&= \{[(\{\bullet\},\emptyset,\{\bullet\mapsto pq\dagger t_i\})\cdot X] \mid [X]\in[\![L_i]\!]_{\text{SOP}} \text{ and } 1\le i\le n\} \\
[\![L_1 \parallel L_2]\!]_{\text{SOP}} &= \{[X_1 \parallel X_2] \mid [X_1]\in[\![L_1]\!]_{\text{SOP}} \text{ and } [X_2]\in[\![L_2]\!]_{\text{SOP}}\} \\
[\![L_1 \cdot L_2]\!]_{\text{SOP}} &= \{[X_1 \cdot X_2] \mid [X_1]\in[\![L_1]\!]_{\text{SOP}} \text{ and } [X_2]\in[\![L_2]\!]_{\text{SOP}}\}
\end{aligned}
$$

Figure 10 Interpretation of local types as SOPs.

actions are performed, a commitment is gradually made towards a subset of behaviours that are still "eligible". The transition system generated by rule [SopA] is trace equivalent to the transition system generated by rule [SopB]. However, these two transition system are *not* bisimulation equivalent: the former can be simulated by the latter, but not vice versa (i.e., rule [SopB] subsumes rule [SopA]). Shortly, we will argue that late choices (i.e., rule [SopB]) are appropriate in our setting, so we will use rule [SopB] instead of rule [SopA].

Interpretation. Let $[\![L]\!]_{\text{SOP}}$ denote the *interpretation* of local type L as a SOP. Formally, $[\![\text{-}]\!]_{\text{SOP}} : \mathbb{L} \to 2^{\mathbf{Pom}}$ is the smallest function induced by the equations in Figure 10. The interpretation of **end** is the SOP that contains only the empty pomset. The interpretation of $pq\dagger\{t_i.L_i\}_{1\le i\le n}$ is the SOP that contains for every $1\le i\le n$, and for every pomset X in the interpretation of L_i, the pomset in which a $pq\dagger t_i$-labelled event precedes all events in X; in the visualisation, the arrow represents precedence ($\prec$). The interpretations of $L_1 \parallel L_2$ and $L_1 \cdot L_2$ are the pairwise parallel composition and the pairwise sequential composition.

▶ **Example 13.** The following SOP is the interpretation of $L_{\mathtt{s}}$ in Example 3:

$$[\![L_{\mathtt{s}}]\!]_{\text{SOP}} = [\![(\mathtt{sb}!\mathtt{Descr} \parallel \mathtt{sb}!\mathtt{Price})\cdot \mathtt{sb}?\{\mathtt{Acc},\mathtt{Rej}\}]\!]_{\text{SOP}}$$

$$= \left\{ \left[\begin{matrix} \overset{\mathtt{sb}!\mathtt{Descr}}{\bullet} \searrow \\ \overset{\mathtt{sb}!\mathtt{Price}}{\bullet} \nearrow \end{matrix} \overset{\mathtt{bs}?\mathtt{Acc}}{\bullet} \right], \left[\begin{matrix} \overset{\mathtt{sb}!\mathtt{Descr}}{\bullet} \searrow \\ \overset{\mathtt{sb}!\mathtt{Price}}{\bullet} \nearrow \end{matrix} \overset{\mathtt{bs}?\mathtt{Rej}}{\bullet} \right] \right\}$$ ⌟

▶ **Example 14.** The following SOP is the interpretation of $L_{\mathtt{m}}$ in Example 4:

$$[\![L_{\mathtt{m}}]\!]_{\text{SOP}} = [\![\mathtt{mw}_1!\mathtt{Work}\,.\,\mathtt{mw}_2!\mathtt{Work}\,.\,(\mathtt{w}_1\mathtt{m}?\mathtt{Done} \parallel \mathtt{w}_2\mathtt{m}?\mathtt{Done})]\!]_{\text{SOP}}$$

$$= \left\{ \left[\overset{\mathtt{mw}_1!\mathtt{Work}}{\bullet} \longrightarrow \overset{\mathtt{mw}_2!\mathtt{Work}}{\bullet} \begin{matrix} \nearrow \overset{\mathtt{w}_1\mathtt{m}?\mathtt{Done}}{\bullet} \\ \searrow \overset{\mathtt{w}_2\mathtt{m}?\mathtt{Done}}{\bullet} \end{matrix} \right] \right\}$$ ⌟

▶ **Example 15.** The following SOP is the interpretation of $L_{\mathtt{m}}$ in Example 12:

$$[\![L_{\mathtt{m}}]\!]_{\text{SOP}} = [\![\mathtt{mw}_1!\mathtt{Work}\,.\,\mathtt{mw}_2!\mathtt{Work}\,.\,\mathtt{mw}_3!\mathtt{Work}\,.\,(\mathtt{w}_1\mathtt{m}?\mathtt{Done} \parallel \mathtt{w}_2\mathtt{m}?\mathtt{Done} \parallel \mathtt{w}_3\mathtt{m}?\mathtt{Done})]\!]_{\text{SOP}}$$

$$= \left\{ \left[\overset{\mathtt{mw}_1!\mathtt{Work}}{\bullet} \longrightarrow \overset{\mathtt{mw}_2!\mathtt{Work}}{\bullet} \longrightarrow \overset{\mathtt{mw}_3!\mathtt{Work}}{\bullet} \begin{matrix} \nearrow \overset{\mathtt{w}_1\mathtt{m}?\mathtt{Done}}{\bullet} \\ \rightarrow \overset{\mathtt{w}_2\mathtt{m}?\mathtt{Done}}{\bullet} \\ \searrow \overset{\mathtt{w}_3\mathtt{m}?\mathtt{Done}}{\bullet} \end{matrix} \right] \right\}$$

(We explain the meaning of the dashed lines later.)

This SOP avoids complication 1 in Figure 5 (i.e., state explosion): it has linearly many events in the number of unordered receives (cf. the DFA in Example 12). ⌟

4.2 From SOPs to APIs

To encode SOPs as APIs, the key insight is that an input-enabled process P is well-typed by local type L (§ 2) if, and only if, every possible sequence of sends and receives by P precisely *covers* the set of events of a pomset $\mathcal{X}$ in $[\![L]\!]_{\text{SOP}}$ (i.e., every send or receive in the sequence is an event of $\mathcal{X}$, and vice versa) and *respects* the precedence relation (i.e., the sequence is a linearisation of $\mathcal{X}$). That is, the sequence of sends and receives by P must correspond to a sequence of transitions of the SOP, derived using the rules in Figure 9. The "trick", then, is to structure the API in such a way that when the compiler successfully type-checks the API's usage, it has effectively validated coverage and respectfulness. In the rest of this subsection, we show how to achieve this in Scala 3 using match types.

We proceed in three paragraphs: first, to set the stage, we present a basic encoding of SOPs for *choice-free protocols*; next, we extend the basic encoding with advanced support for *concurrent subprotocols;* last, we also extend the basic encoding with advanced support for *choice-based protocols*. We note that "advanced" pertains to the encodings, but not to the features (i.e., choices are a basic feature in theory, but when modelled as SOPs, they require an advanced encoding in practice). Furthermore, we note that the two extensions cannot be used together yet; we explain the challenges to combine them at the end of this section.

Our aim in this subsection is to convey the main ideas and insights of the encoding as clearly as possible. To this end, we focus mostly on examples (instead of presenting the general encoding schemes). We do emphasise upfront, though, that the encodings are general, as also evidenced by our tool and the examples that we distribute with it (§ 5).

Basic encoding: choice-free protocols. The idea behind the basic encoding of singleton SOPs is similar to the states-as-type-parameters encoding of DFAs. That is, as in § 3.3, a single generic state class is generated, combined with the usage of match types $\langle r\rangle$`$SendReturn` and $\langle r\rangle$`$RecvArgument` for the return and argument values of methods `send` and `recv`:

```
class ⟨r⟩$State[X](x: X, net: Network) extends UseOnce:
  def send[Q, E](q: Q, e: E): ⟨r⟩$SendReturn[X, Q, E] = { use; ... }
  def recv(fff: ⟨r⟩$RecvArgument[X]): ⟨r⟩$Final = { use; ... }
```

The main difference is the way in which the type parameter of $\langle r\rangle$`$State` is instantiated: whereas `N` in § 3.3 was instantiated with numeric literal types to identify states in a DFA, `X` in this section is instantiated with tuples of *boolean literal types* (e.g., in Scala, symbol "`true`" denotes both value true and a type with value true as its only inhabitant) to represent events in the current pomset. For instance, if $\{1, 2, 3\}$ is the set of events, then its representation as a tuple is `(v1, v2, v3)`, where each of `v1`, `v2`, and `v3` is either type `true` or type `false`. Intuitively, if an event is represented as `true`, then it is still *enabled* (i.e., it has not happened yet); if it is represented as `false`, then it is *disabled* (i.e., it has happened already). We note that "state" in this section should be understood as "the current pomsets in the SOP".

Match type $\langle r\rangle$`$SendReturn` has three type parameters, called `X`, `Q`, and `E`, which represent the current pomset $\mathcal{X}$, the receiver q, and the type t of the value to send. If the pomset has a send transition $\mathcal{X} \xrightarrow{rq!t} \mathcal{X}'$ (derived using the rules in Figure 9 through a new auxiliary match type $\langle r\rangle$`$Pom$Send`), then the compiler succeeds to reduce $\langle r\rangle$`$SendReturn[`$\langle\mathcal{X}\rangle$`,`$\langle q\rangle$`,`$\langle t\rangle$`]` to $\langle r\rangle$`$State[`$\langle\mathcal{X}'\rangle$`]`; this is the type of the fresh successor state object after sending. Match type $\langle r\rangle$`$RecvArgument` has one type parameter, called `X`, which represents the current pomset $\mathcal{X}$. If the pomset has receive transitions $\mathcal{X} \xrightarrow{p_1 r?t_1} \mathcal{X}'_1, \ldots, \mathcal{X} \xrightarrow{p_n r?t_n} \mathcal{X}'_n$ (derived using the rules in Figure 9 through a new auxiliary match type $\langle r\rangle$`$Pom$Recv`), then the compiler succeeds to reduce $\langle r\rangle$`$RecvArgument[`$\langle\mathcal{X}\rangle$`]` to a tuple of function types each of which is of the form `(`$\langle p_i\rangle$`,`$\langle t_i\rangle$`,`$\langle r\rangle$`$State[`$\langle\mathcal{X}'\rangle$`]) =>` $\langle r\rangle$`$Final`; these are the types of the continuations after receiving. The following examples demonstrate these match types.

▶ **Example 16.** The following API (excerpt) is the encoding of $[\![L_{\mathtt{m}}]\!]_{\mathrm{SOP}}$ in Example 14. We present it in two steps. First, we define auxiliary match types `M$Pom$Send` and `M$Pom$Recv` to derive transitions of the single pomset in $[\![L_{\mathtt{m}}]\!]_{\mathrm{SOP}}$ using rules [LPO] and [POM] in Figure 9. Second, we use these auxiliary match types to define `M$SendReturn` and `M$RecvArgument`.

1. The first listing shows the two auxiliary match types:

```
type M$Pom$Send[X, Q, E] = (X, Q, E) match
  case ((true, v2, v3, v4), W1, Work)    => (false, v2, v3, v4)
  case ((false, true, v3, v4), W2, Work) => (false, false, v3, v4)
  case Any                               => Error

type M$Pom$Recv[X, P, E] = (X, P, E) match
  case ((v1, false, true, v4), W1, Done) => (v1, false, false, v4)
  case ((v1, false, v3, true), W2, Done) => (v1, false, v3, false)
  case Any                               => Error
```

   Every case in match type `M$Pom$Send` encodes a send transition of the pomset represented by `X`, with `Q` as the receiver and `E` as the type of the value to send, derived using rules [LPO] and [POM] in Figure 9. The first case states that if `X` matches `(true, v2, v3, v4)`, where `v2`, `v3`, and `v4` are local type variables that are bound by matching (i.e., the first event in the pomset has not yet happened and is still enabled, hence `true`; the other events are irrelevant), and if `Q` and `E` match `W1` and `Work`, then the pomset can make a transition to `(false, v2, v3, v4)` when work is sent to worker 1 (i.e., the first event has happened and is disabled, hence `false`; the other events are unaffected). We note that it is more convenient to "set" events to `false` instead of removing them as in rule [LPO]. The second case is similar, except that it imposes a precedence constraint: to match `X`, its first element must be `false`. That is, for the second event to happen, the first event must have already happened (i.e., the second event must have become minimal to satisfy the premise of rule [LPO]). The third case states that if the first two cases do not apply, then a send of a value of type `E` to receiver `Q` cannot happen in `X`. Here, `Error` is a special type that we use to explicitly represent "failed reduction"; it is dealt with in the next step. Match type `M$Pom$Recv` is similar, but for receives instead of sends.

2. The second listing shows $\langle r\rangle$`$SendReturn` and $\langle r\rangle$`$RecvArgument`:

```
type M$SendReturn[X, Q, E] =            /* |-then-| */
  IfThenElse[IsError[M$Pom$Send[X, Q, E]], Unit, M$State[M$Pom$Send[X, Q, E]]]
         /* |---if-----------------------|       |---else--------------------| */

type M$RecvArgument[X] =
  Simplify[(IfThenElse[IsError[M$Pom$Recv[X, W1, Done]], Unit,
                       (W1, Done, M$State[M$Pom$Recv[X, W1, Done]]) => M$Final],
            IfThenElse[IsError[M$Pom$Recv[X, W2, Done]], Unit,
                       (W2, Done, M$State[M$Pom$Recv[X, W2, Done]]) => M$Final])]
```

   To reduce match type `M$SendReturn[X, Q, E]`, the compiler checks if `M$Pom$Send[X, Q, E]` reduces to `Error`, using "utility match types" `IfThenElse` and `IsError`. If it does (i.e., a send of `E` to `Q` in `X` cannot happen), then the type of the fresh successor state object is `Unit` (i.e., to indicate that something is wrong). In contrast, if it does not reduce to `Error` (i.e., the send can happen), then the type of the fresh successor state is proper.
   To reduce match type `M$RecvArgument[X]`, the compiler uses a similar approach to determine for every possible receive (characterised in terms of the sender and the type of the value) if it can happen or not in `X`. The result is a tuple that consists of either a proper continuation function or `Unit` for every possible receive; the `Unit`s are subsequently removed using utility match type `Simplify`.

We note that exactly the same function that implements the master in Example 9 is also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/3PyXcwBKS2argRL9oPlspg`. ◀

Advanced encoding: concurrent subprotocols. In this paragraph, we extend the basic encoding with an advanced feature that significantly subsumes DFA-based API generation: the ability to *fork* and *join* pomsets. Intuitively, when a fork is performed, the pomset is "broken" into "shards", each of which can evolve independently of the others; when a join is performed, shards are "glued" back together. In terms of typing rules, intuitively, forking corresponds to rule [PAR] in Figure 7, while joining corresponds to rule [SEQ]. In the DFA encodings, rule [PAR] has no corresponding representation, as the parallel structure is lost in translation. In contrast, in the SOP encodings, the parallel structure is still there and thus can be exploited. This extension is crucial to effectively support concurrent subprotocols (i.e., avoid branch explosion; Figure 5), as will be demonstrated shortly in Example 18.

A first essential ingredient that we need, is an analysis procedure to statically identify shards. To explain it, suppose that $[X] = [(V, \prec, \lambda)]$ is a pomset. The idea is to partition V into two kinds of subsets, each of which forms a shard:

Join shards: If $v \in V$, and if $|\mathsf{pred}_X(v)| > 1$, then $\{v\}$ forms a *join shard*.

Sequential shards: If $v \preceq v_1 \prec \cdots \prec v_n \preceq v'$, and if $|\mathsf{succ}_X(v_i) \cup \mathsf{pred}_X(v_i)| \leq 2$, for every $1 \leq i \leq n-1$ (i.e., $v_1, \ldots, v_n$ is a chain), and if either $|\mathsf{succ}_X(v)| > 1$ or $|\mathsf{pred}_X(v_1)| \neq 1$ (i.e., either v_1 is preceded by "fork event" v, or it is an "initial event" itself), and if either $|\mathsf{pred}_X(v') > 1|$ or $|\mathsf{succ}_X(v_n)| \neq 1$ (i.e., v_n is succeeded by "join event" v', or it is a "final event" of "fork event" itself), then $\{v_1, \ldots, v_n\}$ forms a *sequential shard.* That is, a sequential shard is a longest chain of events, optionally preceded by a fork event (if v_1 is not initial), and optionally succeeded by a join event (if v_n is not final or fork).

For instance, the dashed lines in Example 15 visualise four shards. We note that identification of shards is computationally easy: it can be done in polynomial time (in the size of the pomset), using standard graph traversal techniques.

Using this concept of shards, forking and joining generally works as follows:

- If an event e has multiple immediate successors, and if every immediate predecessor has already happened, and if every immediate successor has not yet happened, then the "old pomset" can be forked into "new sub-pomsets" by breaking it into shards: for every immediate successor of e, there is a new sub-pomset that consists of all sequential shards that are reachable from e without passing through a join shard; the join shards and all shards beyond are temporarily disabled. That is, each new sub-pomset can evolve independently within the boundaries of its shards, but to go further, a join is needed first. We also note that sub-pomsets can be recursively forked.
- If an event e has multiple immediate predecessors, and if every immediate predecessor has already happened, then the sub-pomsets can be joined by gluing their shards, re-enabling the temporarily disabled join shards and all shards beyond.

To incorporate these concepts, we extend the basic encoding as follows:

- Class $\langle r \rangle$`$State` is extended with a method `fork`. At execution time, when `fork` is called, a fork is performed and fresh successor state objects are returned; they can be used independently of each other (i.e., concurrent subprotocols). At compile-time, to check that this method call is actually allowed in the current pomset, the compiler tries to reduce a match type: if it succeeds, the call is allowed; if it fails, it is not. More precisely, the idea is to use a match type $\langle r \rangle$`$ForkReturn` for the *return value*; it has one type parameter, called `X`, which represents the current pomset $\mathcal{X}$. If $\mathcal{X}$ can be forked into

sub-pomsets $\mathcal{X}_1, \ldots, \mathcal{X}_n$, then the compiler succeeds to reduce $\langle r \rangle$`$ForkReturn[`$\langle\mathcal{X}\rangle$`]` to a tuple `(`$\langle r \rangle$`$State[`$\langle\mathcal{X}_1\rangle$`],...,`$\langle r \rangle$`$State[`$\langle\mathcal{X}_n\rangle$`])`; these are the types of the fresh successor state objects after forking. In contrast, if $\mathcal{X}$ cannot be forked, then the compiler fails to reduce and yields an error. Thus, `fork` looks as follows:

```
def fork(): M$ForkReturn[X] = { use; ... }
```

We note that forking a pomset also counts as "usage": after calling `fork`, the state object should not be used again to send or receive.

- Class $\langle r \rangle$`$State` is also extended with a static method `join` for every "join event" in the pomset, using overloading. At execution time, when `join` is called, multiple final state objects for sub-pomsets are collected (passed as arguments) and a single fresh successor state object is returned. At compile-time, to check that this method call is actually allowed, the compiler checks if the types of the actual parameters match the types of the formal parameters of one of the overloaded `join` methods.
- To conveniently distinguish sub-pomsets, the representation `X` of the current pomset (i.e., tuple of boolean literal types) is extended with an extra element, namely a *fork identifier* (i.e., numeric literal type): if the current pomset was previously forked off (i.e., `X` actually represents a sub-pomset), then the fork identifier is a numeric literal type that identifies the immediate successor of the "fork event"; else, the fork identifier is `0`.
- To enable method `recv` in class $\langle r \rangle$`$State` to return different final states for different sub-pomsets, the type of its return value is refined to depend on fork identifiers. The idea is to use an auxiliary match type $\langle r \rangle$`$Pom$Final` to derive final states: it has one type parameter, called `X`, which represents the current sub-pomset. Based on the fork identifier in `X`, the compiler reduces $\langle r \rangle$`$Pom$Final[X]` to a tuple in which all events of the enabled shards are represented as `false` (i.e., they have happened). Thus, `recv` looks as follows:

```
def recv(fff: ⟨r⟩$RecvArgument[X]): ⟨r⟩$State[⟨r⟩$Pom$Final[X]] = { use; ... }
```

The following examples demonstrate these match types.

▶ **Example 17.** The following API (excerpt) is the encoding of $[\![L_{\mathtt{m}}]\!]_{\text{SOP}}$ in Example 14. We present it in three steps. First, we define auxiliary match types `M$Pom$Send` and `M$Pom$Recv` (similar to Example 16) and auxiliary match type `M$Pom$Final` (new). Second, we use these auxiliary match types to define `M$SendReturn` and `M$RecvArgument` (similar to Example 16) and `M$Fork$Return` (new). Third, we define static method `join`.

1. The first listing shows the three auxiliary match types:

```
type M$Pom$Send[X, Q, E] = (X, Q, E) match
  case ((n, true, v2, v3, v4), W1, Work)    => (n, false, v2, v3, v4)
  case ((n, false, true, v3, v4), W2, Work) => (n, false, false, v3, v4)
  case Any                                  => Error

type M$Pom$Recv[X, P, E] = ...

type M$Pom$Final[X] = X match
  case (0, v1, v2, v3, v4) => (0, false, false, false, false)
  case (3, v1, v2, v3, v4) => (3, v1, v2, false, v4)
  case (4, v1, v2, v3, v4) => (4, v1, v2, v3, false)
```

Match types `M$Pom$Send` and `M$Pom$Recv` are the same as in Example 16, except that tuple `X` has an extra element `n`, namely a numeric fork identifier. Match type `M$Pom$Final` is new: it is used to infer when a sub-pomset has fully evolved (i.e., within the boundaries of its shards, but not beyond). For instance, the second case states that the sub-pomset identified by `3` is final when the event identified by `3` has happened.

2. The second listing shows `M$SendReturn`, `M$RecvArgument`, and `M$Fork$Return`:

```
type M$SendReturn[X, Q, E] = ...

type M$RecvArgument[X] =
  Simplify[(IfThenElse[IsError[M$Pom$Recv[X, W1, Done]], Unit,
                      (W1, Done, M$State[M$Pom$Recv[X, W1, Done]]) =>
                        M$State[M$Pom$Final[X]]],
            ...)]

type M$ForkReturn[X] = X match
  case (0, v1, v2, true, true) => (
    M$State[(3, v1, v2, true, false)], M$State[(4, v1, v2, false, true)])
```

   Match types `M$SendReturn` and `M$RecvArgument` are the same as in Example 16, except that the return types of the continuations as computed by `M$RecvArgument` depend on the fork identifier in `X`, using match type `M$Pom$Final`. Match type `M$ForkReturn` is new: it is used to infer which old pomsets can be forked into which tuples of new sub-pomsets. In this example, there is only one case. It states that if the old pomset is unforked, and if the third and fourth events have not yet happened, then it can be forked into two new sub-pomsets: one for the third event and one for the fourth event.

   In general, match type $\langle r \rangle$`$ForkReturn` has as many cases as there are "fork events" in the pomset. That is, we allow a pomset to be forked only right after a fork event, before any event of the immediate successors has happened (if we would allow it also "later", then we would need to generate exponentially many cases).

3. The third listing shows `join`:

```
object M$State: // static methods of class M$State
  def join(
    s1: M$State[(3, false, false, false, false)],
    s2: M$State[(4, false, false, false, false)]
  ): M$State[(0, false, false, false, false)] = ...
```

   Method `join` is needed in this example to provide a final state object with a unique type; it is implicitly present in every pomset with more than one maximal element.

   In general, $\langle r \rangle$`$State` has as many `join` methods as there are "join events" in the pomset.

We note that exactly the same function that implements the master in Example 9 is also well-typed using the API in this example. Thus, the fork–join extension of the basic encoding is backwards-compatible. To additionally demonstrate forking and joining, the following well-typed function implements the master as well:

```
def master(s: M$Initial): M$Final =
  val (s1, s2) = s.send(W1, new Work).send(W2, new Work).fork()
  val f1 = Future { s1.recv((_, _, s) => { println("#1"); s }) }
  val f2 = Future { s2.recv((_, _, s) => { println("#2"); s }) }
  Await.result(for { t1 <- f1; t2 <- f2 } yield M$State.join(t1, t2), ...)
```

In the first line of the body, the two sends are sequentially performed as before; after that, the pomset is forked into two pomsets, divided over successor state objects `s1` and `s2`. On the second and third line, the two receives are performed concurrently using two *futures* (built-in Scala mechanism for asynchronous programming, using a default thread pool). On the fourth line, the results of the futures are awaited. Any change in the order of the actions (sends, receives, fork, join) results in a compile-time error. For details, see: `https://scastie.scala-lang.org/RoIs43OcTsS3w9wh8GC74w`. ⌟

▶ **Example 18.** The following well-typed function implements the master (with three workers), using an API that encodes the SOP in Example 15, including the fork–join extension.

```
def master(s: M$Initial): M$Final =
  val (s1, s2, s3) = s.send(W1, new Work).send(W2, new Work).send(W3, new Work).fork()
  val f1 = Future { s1.recv((_, _, s) => { println("#1"); s }) }
  val f2 = Future { s2.recv((_, _, s) => { println("#2"); s }) }
  val f3 = Future { s3.recv((_, _, s) => { println("#3"); s }) }
  Await.result(
    for { t1 <- f1; t2 <- f2; t3 <- f3 } yield M$State.join(t1, t2, t3), ...)
```

This implementation avoids complication 2 in Figure 5 (i.e., branch explosion): it has linearly many branches in the number of unordered receives (cf. the implementation in Example 12, which has exponentially many branches). For details, see: `https://scastie.scala-lang.org/MUEFr4ZvSEyFepAJWAKh3g`. ⌟

Advanced encoding: choice-based protocols. In the previous paragraphs, we presented the basic encoding of singleton SOPs and the fork–join extension. In this last paragraph, we present another extension of the basic encoding to support non-singleton SOPs.

Intuitively, a non-singleton SOP $\{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$ represents a *choice* among n possible local behaviours. The trouble is that both at compile-time and initially at execution-time, it is still unknown which of the n pomsets will actually be chosen (e.g., the seller in the seller-buyer protocol initially does not know yet if the buyer will accept or reject the offer). In particular, this observation entails that we cannot "compositionally" apply the basic encoding to each of the n pomsets and require a process to choose one of them in the beginning; generally, there is no way in which the process can make such an early choice upfront. That is, rule [SopA] in Figure 9, which formalises the idea of early choices, is too inflexible.

Instead, a process needs to keep its options open for as long as possible. To achieve this, in accordance with rule [SopB] in Figure 9, which formalises the idea of late choices, the plan is to incrementally refine the set of "eligible pomsets" (to become the chosen one), by accumulating knowledge during the execution of the process. That is (cf. rule [SopB]), initially, all pomsets are eligible; subsequently, every time a send or receive happens, all eligible pomsets *that do not allow the action to happen*, become ineligible. In this way, when the process terminates, coverage and respectfulness are satisfied if, and only if, all events of at least one remaining eligible pomset have happened. It is straightforward to perform such an incremental eligibility analysis dynamically; the challenge is to "trick" the compiler into doing it statically. To achieve this, we extend the basic encoding as follows:

- Class $\langle r \rangle$`$State` is extended with additional type parameters: instead of just `X`, which represents the one pomsets in a singleton SOP, it has `X1,...,Xn`, which represent the n pomsets in a non-singleton SOP.
- Similarly, match types $\langle r \rangle$`$SendReturn` and $\langle r \rangle$`$RecvArgument` are extended with additional type parameters `X1,...,Xn`. Furthermore, the definitions of these match types are extended to derive send and receive transitions of the non-singleton SOP represented by `X1,...,Xn` using rule [SopB] in Figure 9. We note that, essentially, the basic encoding of singleton SOPs is a special case of the advanced encoding of non-singleton SOPs.

The following example demonstrates these match types.

▶ **Example 19.** The following API (excerpt) is the encoding of $[\![L_{\mathtt{s}}]\!]_{\text{SOP}}$ in Example 13. We present it in two steps. First, we define auxiliary match types `S$Pom1$Send` and `S$Pom1$Recv` to derive transitions of the "left" pomset in $[\![L_{\mathtt{s}}]\!]_{\text{SOP}}$, and auxiliary match types `S$Pom2$Send` and `S$Pom2$Recv` to derive transitions of the "right" pomset in $[\![L_{\mathtt{s}}]\!]_{\text{SOP}}$, using rules [Lpo] and [Pom] in Figure 9 (similar to Example 16). Second, we use these auxiliary match types to define `S$SendReturn` and `S$RecvArgument`.

1. The first listing shows the four auxiliary match types:

```
type S$Pom1$Send[X, Q, E] = (X, Q, E) match
  case ((true, v2, v3), B, Descr) => (false, v2, v3)
  case ((v1, true, v3), B, Price) => (v1, false, v3)
  case Any                        => Error

type S$Pom1$Recv[X, Q, E] = (X, P, E) match
  case ((false, false, true), B, Acc) => (false, false, false)
  case Any                            => Error

type S$Pom2$Send[X, Q, E] = (X, Q, E) match
  case ((true, v2, v3), B, Descr) => (false, v2, v3)
  case ((v1, true, v3), B, Price) => (v1, false, v3)
  case Any                        => Error

type S$Pom2$Recv[X, P, E] = (X, P, E) match
  case ((false, false, true), B, Rej) => (false, false, false)
  case Any                            => Error
```

   Conceptually, there is nothing new here relative to the basic encoding, except that we should be more precise about the meaning of `Error`: it stands for "ineligible".

2. The second listing shows `S$SendReturn` and `S$RecvArgument`:

```
type S$SendReturn[X1, X2, Q, E] =
  IfThenElse[
    And[IsError[S$Pom1$Send[X1, Q, E]], IsError[S$Pom2$Send[X2, Q, E]]], Unit,
    S$State[S$Pom1$Send[X1, Q, E], S$Pom2$Send[X2, Q, E]]]

type S$RecvArgument[X1, X2] =
  Simplify[(
    IfThenElse[
      And[IsError[S$Pom1$Recv[X1, B, Acc]], IsError[S$Pom2$Recv[X2, B, Acc]]], Unit,
      (B, Acc, S$State[S$Pom1$Recv[X1, B, Acc], S$Pom2$Recv[X2, B, Acc]]) =>
        S$Final],
    ...)]
```

   To reduce match type `S$SendReturn[X1, X2, Q, E]`, where `X1` and `X2` represent the two pomsets in the SOP, the compiler checks if `M$Pom1$Send[X1, Q, E]` and `M$Pom2$Send[X2, Q, E]` reduce to `Error`. If they do (i.e., a send of `E` to `Q` can happen neither in `X1` nor in `X2`), then the type of the fresh successor state object is `Unit` (i.e., all pomsets have become ineligible). In contrast, if they do not both reduce to `Error` (i.e., the send can happen in `X1`, or in `X2`, or in both), then the type of the fresh successor state is proper; it is formed by evolving both `X1` and `X2`. There are three cases:

   - If `X1` and `X2` are bound to a non-`Error` type, and they evolve to non-`Error` types, then the corresponding type parameters of the successor remain bound to non-`Error` types. That is, both pomsets remain eligible.
   - If `X1` (resp. `X2`) is bound to a non-`Error` type, but it evolves to `Error`, then the corresponding type parameter of the successor becomes bound to `Error` as well. That is, the first pomset (resp. second pomset) becomes ineligible. We note that it is more convenient to "set" pomsets to `Error` instead of removing them as in rule [SopB].
   - If `X1` (resp. `X2`) is bound to `Error`, then it "evolves" again to `Error` (see previous listing), so the corresponding type parameter of the successor remains bound to `Error` as well. That is, the first pomset (resp. second pomset) remains ineligible.

   To reduce match type `M$RecvArgument[X1, X2]`, the compiler uses a similar approach to determine for every possible receive if it can happen or not in `X1` and `X2`.

We note that exactly the same function that implements the master in Example 8 is also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/WN5ZmEMcRh2ecUMaCgvjPA`. ⌟

It remains a largely open question how to use both the fork–join extension of the previous paragraph, and the choice extension of this paragraph, together. The only situation in which we know how to do it, is when a non-singleton SOP has evolved in such a way that only one of its pomsets has remained eligible; in that case, it has effectively become a singleton SOP, to which the fork–join extension readily applies. The main challenge to also support forking and joining of SOPs with multiple eligible pomsets is that we need to devise a mechanism to control non-forked pomsets. For instance, suppose that we have a SOP $\{[X_1], [X_2]\}$ (both eligible), and suppose that $[X_1]$ can be forked into $[X_1^\dagger]$ and $[X_1^\ddagger]$. By naively performing the fork, we get two fresh successor state objects: one for $\{[X_1^\dagger], [X_2]\}$ and one for $\{[X_1^\ddagger], [X_2]\}$. However, without an additional control mechanism, the events in $[X_2]$ are now allowed to be executed *twice*. Solving this non-trivial problem is part of future work.

5 Tool Support: Pompset

We developed a tool, called Pompset (portmanteau of `pom_set` and `__mps_t`), to automatically generate SOP-based APIs, according to the workflow in Figure 4. More precisely, Pompset consumes a global type as input and produces a set of APIs as output as follows:

1. **From global type to local types:** First, the global type is parsed to a "global AST". Next, for every role that occurs in the global AST, the global AST is projected to a "local AST" in accordance with § 2.

2. **From local types as SOPs:** Every local AST is interpreted to a "SOP data structure" in accordance with § 4.1.

3. **From SOPs to APIs:** Every SOP data structure is encoded as an API in accordance with § 4.2, including the ability to fork–join SOPs. (For practical/engineering reasons, though, the generated code is not completely identical, but it follows the same ideas and insights and works morally the same as in the examples.)

   Technically, the encoding is carried out by filling generic templates with specific data from the SOP under consideration. We refer to our artefact (published in DARTS), for details of the templates and the filling process; it includes source code and build instructions. Furthermore, it includes additional examples to demonstrate the generality of the implemented encoding scheme.

   We note that generated APIs also consist of functions to spawn processes and transparently set up the underlying communication infrastructure (i.e., transport abstraction). The latter is based on shared-memory channels, but it could work equally well with TCP channels; it just requires additional engineering effort.

The guarantees that APIs generated by Pompset provide at compile-time, are as usual (§ 3.2): deadlock freedom and protocol compliance, modulo non-linear usage of state objects (checked at execution-time), and modulo uncontrollable sources of non-terminating/ exceptional behaviour. In addition to API generation as presented so far, Pompset also offers *enhanced error messages* and *additional pomset support* to improve both the usability of the generated APIs and the usefulness of Pompset; we describe these features in [8, § A].

Pompset is written in Scala, open source, and it has a browser-based graphical user interface; we provide a screenshot in [8, § B].

6 Conclusion

In this paper (pearl), we revisited and revised the API generation approach to support the MPST method in practice. Regarding the "revisitation", using Scala 3, we presented two versions of the existing DFA-based API generation: states-as-classes (existing) and states-as-type-parameters (new, by leveraging match types in Scala 3). Regarding the "revision", we presented a new SOP-based API generation (again, by leveraging match types in Scala 3). Through this fresh perspective, we showed how to effectively support concurrent subprotocols for the first time in MPST practice. The SOP-based version is incorporated in a new tool.

Regarding choices, DFA-based and SOP-based API generation are equally expressive. However, the DFA approach supports loops, which the SOP approach currently does not. In contrast, the SOP approach supports forking/joining of subprotocols, which the DFA approach does not. Thus, for now, a trade-off needs to be made when choosing which encoding to use, but our vision is that the SOP approach has the potential to subsume the DFA approach (when the type system of the host language supports a kind of match types).

6.1 Related Work

Local types as DFAs. The idea to interpret local types as DFAs was conceived by Deniélou and Yoshida [11,12], within the framework of *communicating finite state machines* (CFSM) [5]. A central notion in this work is *multiparty compability*: it is used to provide a sound and complete characterisation between global types and *systems* (i.e., parallel compositions of DFAs that communicate through asynchronous channels). Multiparty compatibility was further studied and generalised in subsequent work, to cover timed behaviour [4], more flexible choice [28], and non-synchronisability [29].

DFAs as APIs. The idea to encode DFAs as APIs was conceived by Hu and Yoshida [20,21], for Java. The approach has subsequently been used in combination with numerous other programming languages as well, including F# [33], F* [46], Go [7], OCaml [45], PureScript [25], Rust [27], Scala [37], and TypeScript [31]. In many of these works, distinguished capabilities of the type system of "the host" are leveraged to offer additional compile-time guarantees and/or support MPST extensions. For instance, Neykova et al. and Zhou et al. use type providers in F# and refinement types in F* to generate APIs that support MPST-based refinement [33,46], while King et al. and Lagaillardie et al. use indexed monads in PureScript and ownership types in Rust to support static linearity [25,27].

Alternative approaches (i.e., not based on API generation) to apply the MPST method in combination with mainstream programming languages include the work of Imai et al. [22] (for OCaml), the work of Harvey et al., Kouzapas et al., and Voinea et al. [17,26,44] (for Java, using a typestate extension), and the work of Scalas et al. [40,41] (for Scala, using an external model checker). Furthermore, there exist approaches to apply the MPST method that rely on *monitoring* and/or *assertion checking* at execution-time [2,3,9,16,32,33]. The motivation is that in practice, some distributed components of a system might not be amenable to static type-checking (e.g., the source code is unavailable), but they can be dynamically monitored for compliance.

Local types as pomsets. The idea to encode local types as pomsets was conceived by Guanciale and Tuosto [14], in a continuation of earlier work on pomset-based semantics of global types [43]. A key contribution of Guanciale and Tuosto is a sound and complete procedure to determine if a SOP-interpretation of a global type is *realisable* as a collection

of SOP-interpretations of the global type's projections; most procedures in the MPST literature are only sound. The PomCho tool [15] supports analysis (including counterexample generation), visualisation, and projection of pomsets. The crucial difference with our tool is that PomCho cannot generate APIs.

6.2 Future Work

We intend to demonstrate the potential of SOP-based API generation with this paper, and we believe that it can become the starting point for many improvements to the current design. Concretely, we are pursuing two pieces of future work.

First, as explained at the end of § 4.2, fork–join support and choice support can be used together only to a limited extent. We are currently approaching this open problem from two angles: on the practical side, we try to devise a mechanism to control non-forked pomsets, without changing the underlying foundations; meanwhile, on the theoretical side, we are studying an alternative pomset-based version of MPST theory that should make choices simpler to support (inspired by *branching automata* [30] and *pomset automata* [24]).

Second, our current version of SOP-based API generation does not support loops. This open problem is foundational: in the same way that Kleene star gives rise to infinite regular languages of finite words, a looping construct in the grammar of global/local types would give rise to infinite sets of finite pomsets. In theory, this is fine; in practice, it is not (i.e., generated APIs would need to be infinite as well). Solving this problem is another reason for us to study an alternative pomset-based version of MPST theory, in which loops can be represented finitely. We expect the key ideas and insights of SOP-based API generation in this paper to remain applicable, though.

References

1 Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proc. ACM Program. Lang.*, 6(POPL):1–24, 2022.

2 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017.

3 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2010.

4 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

5 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

6 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012.

7 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019.

8 Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. Api generation for multiparty session types, revisited and revised using scala 3 (full version). Technical Report OUNL-CS-2022-03, Open University of the Netherlands, 2022.

9 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015.

10 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011.
11 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012.
12 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013.
13 Jay L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61:199–224, 1988.
14 Roberto Guanciale and Emilio Tuosto. Realisability of pomsets. *J. Log. Algebraic Methods Program.*, 108:69–89, 2019.
15 Roberto Guanciale and Emilio Tuosto. Pomcho: A tool chain for choreographic design. *Sci. Comput. Program.*, 202:102535, 2021.
16 Ruben Hamers and Sung-Shik Jongmans. Discourje: Runtime verification of communication protocols in clojure. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 2020.
17 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty session types for safe runtime adaptation in an actor language. In *ECOOP*, volume 194 of *LIPIcs*, pages 10:1–10:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.
19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.
20 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2016.
21 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017.
22 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In *ECOOP*, volume 166 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
23 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 251–279. Springer, 2020.
24 Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi. On series-parallel pomset languages: Rationality, context-freeness and automata. *J. Log. Algebraic Methods Program.*, 103:130–153, 2019.
25 Jonathan King, Nicholas Ng, and Nobuko Yoshida. Multiparty session type-safe web development with static linearity. In *PLACES@ETAPS*, volume 291 of *EPTCS*, pages 35–46, 2019.
26 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Sci. Comput. Program.*, 155:52–75, 2018.
27 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Implementing multiparty session types in rust. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2020.
28 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232. ACM, 2015.
29 Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In *CAV (1)*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019.

30 Kamal Lodaya and Pascal Weil. Series-parallel languages and the bounded-width property. *Theor. Comput. Sci.*, 237(1-2):347–380, 2000.

31 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in typescript with routed multiparty session types. In *CC*, pages 94–106. ACM, 2021.

32 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.*, 29(5):877–910, 2017.

33 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In *CC*, pages 128–138. ACM, 2018.

34 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *CC*, pages 98–108. ACM, 2017.

35 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.

36 Vaughan R. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986.

37 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

38 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

39 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019.

40 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Effpi: verified message-passing programs in dotty. In *SCALA@ECOOP*, pages 27–31. ACM, 2019.

41 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *PLDI*, pages 502–516. ACM, 2019.

42 Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010.

43 Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies. *J. Log. Algebraic Methods Program.*, 95:17–40, 2018.

44 A. Laura Voinea, Ornela Dardha, and Simon J. Gay. Typechecking java protocols with [st]mungo. In *FORTE*, volume 12136 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2020.

45 Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In *FCT*, volume 12867 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2021.

46 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020.

Global Type Inference for Featherweight Generic Java

Andreas Stadelmeier
Duale Hochschule Baden-Württemberg Stuttgart, Campus Horb, Germany

Martin Plümicke
Duale Hochschule Baden-Württemberg Stuttgart, Campus Horb, Germany

Peter Thiemann
Institut für Informatik, Universität Freiburg, Germany

Abstract

Java's type system mostly relies on type checking augmented with local type inference to improve programmer convenience.

We study global type inference for Featherweight Generic Java (FGJ), a functional Java core language. Given generic class headers and field specifications, our inference algorithm infers all method types if classes do not make use of polymorphic recursion. The algorithm is constraint-based and improves on prior work in several respects. Despite the restricted setting, global type inference for FGJ is NP-complete.

2012 ACM Subject Classification Software and its engineering → Language features

Keywords and phrases type inference, Java, subtyping, generics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.28

Related Version *Full Version*: `https://arxiv.org/abs/2205.08768` [28]

Supplementary Material Source code for the accompaning prototype implementation
Software (Source Code): `https://github.com/JanUlrich/FeatherweightTypeInference`
archived at `swh:1:dir:dbbfa3ad5db1b423c098c392af1d4b720db93e0e`
Software (ECOOP 2022 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.8.2.18`

1 Introduction

Java is one of the most important programming languages. In 2019, Java was the second most popular language according to a study based on GitHub data.[1] Estimates for the number of Java programmers range between 7.6 and 9 million.[2] Java has been around since 1995 and progressed through 16 versions.

Swarms of programmers have taken their first steps in Java. Many more have been introduced to object-oriented programming through Java, as it is among the first mainstream languages supporting object-orientation. Java is a class-based language with static single inheritance among classes, hence it has nominal types with a specified subtyping hierarchy. Besides classes there are interfaces to characterize common traits independent of the inheritance hierarchy. Since version J2SE 5.0, the Java language supports F-bounded polymorphism in the form of generics.

[1] `https://www.businessinsider.de/international/the-10-most-popular-programming-languages-according-to-github-2018-10/`

[2] `https://www.zdnet.com/article/programming-languages-python-developers-now-outnumber-java-ones/`, `http://infomory.com/numbers/number-of-java-developers/`

© Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 28; pp. 28:1–28:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
class Pair<X,Y> {
  X fst;
  Y snd;
  Pair<X,Y>(X fst, Y snd) {
    this.fst=fst;
    this.snd=snd;
  }
  <Z> Pair<Z,Y> setfst(Z fst) {
    return new Pair(fst, this.snd);
  }
  Pair<Y,X> swap() {
    return new Pair(this.snd,
        this.fst);
  }
}
```

(a) Featherweight Generic Java (FGJ).

```
class Pair<X,Y> {
  X fst;
  Y snd;
  Pair(fst, snd) {
    this.fst=fst;
    this.snd=snd;
  }
  setfst(fst) {
    return new Pair(fst, this.snd);
  }
  swap() {
    return new Pair(this.snd,
        this.fst);
  }
}
```

(b) FGJ with global type inference (FGJ-GT).

Figure 1 Example code.

Java is generally explicitly typed with some amendments introduced in recent versions. That is, variables, fields, method parameters, and method returns must be adorned with their type. Figure 1a contains a simple example with generics.

While the overhead of explicit types look reasonable in the example, realistic programs often contain variable initializations like the following:[3]

```
HashMap<String, HashMap<String, Object>> outerMap =
  new HashMap<String, HashMap<String, Object>>();
```

Java's *local variable type inference* (since version 10[4]) deals satisfactorily with examples like the initialization of `outerMap`. In many initialization scenarios for local variables, Java infers their type if it is obvious from the context. In the example, we can write

```
var outerMap = new HashMap<String, HashMap<String, Object>>();
```

because the constructor of the map spells out the type in full. More specifically, "obvious" means that the right side of the initialization is

- a constant of known type (e.g., a string),
- a constructor call, or
- a method call (the return type is known from the method signature).

The `var` declaration can also be used for an iteration variable where the type can be obtained from the elements of the container or from the initializer. Alternatively, if the variable is used as the method's return value, its type can be obtained from the current method's signature.

However, there are still many places where the programmer must provide types. In particular, an explicit type must be given for

- a field of a class,
- a local variable without initializer or initialized to `NULL`,
- a method parameter, or
- a method return type.

[3] Taken from `https://stackoverflow.com/questions/4120216/map-of-maps-how-to-keep-the-inner-maps-as-maps/4120268`.

[4] `https://openjdk.java.net/jeps/286`

```
<T> boolean eqPair (Pair<T,T> p){
  return p.fst.equals<T>(p.snd);
}
```

```
eqPair (p) {
  return p.fst.equals(p.snd);
}
```

Figure 2 `eqPair` in FGJ-GT and FGJ.

In this paper, we study *global type inference* for Java. Our aim is to write code that omits most type annotations, except for class headers and field types. Returning to the `Pair` example, it is sufficient to write the code in Figure 1b and global type inference fills in the rest so that the result is equivalent to Figure 1a. Our motivation to study global type inference is threefold.

- Programmers are relieved from writing down obvious types.
- Programmers may write types that leak implementation details. The `outerMap` example provides a good example of this problem. From a software engineering perspective, it would be better to use a more general abstract type like

  ```
  Map<String, Map<String, Object>> outerMap = ...
  ```

  Global type inference finds most general types.
- Programmers may write types that are more specific than necessary instead of using generic types. Here, type inference helps programmers to find the most general type. Suppose we wanted to add a static method `eqPair` for pairs of integers to the `Pair` class.

  ```
  boolean eqPair (Pair<Integer,Integer> p) {
    return p.fst.equals(p.snd);
  }
  ```

  With global type inference it is sufficient to write the code on the right of Figure 2 and obtain the FGJ code with the most general type on the left.

To make our investigation palatable, we focus on global type inference for Featherweight Generic Java [11] (FGJ), a functional Java core language with full support for generics. Our type inference algorithm applies to FGJ programs that specify the full class header and all field types, but omit all method signatures. Given this input, our algorithm infers a set of most general method signatures (parameter types and return types). Inferred types are generic as much as possible and may contain recursive upper bounds.

The inferred signatures have the following round-trip property (relative completeness). If we start with an FGJ program that does not make use of polymorphic recursion (see Section 2.5), strip all types from method signatures, and run the algorithm on the resulting stripped program, then at least one of the inferred typings is equivalent or more general than the types in the original FGJ program.

Contributions

We specify syntax and type system of the language FGJ-GT, which drops all method type annotations from FGJ and the typing of which rules out polymorphic recursion. This language is amenable to polymorphic type inference and each FGJ-GT program can be completed to an FGJ program (see Section 3).

We characterize uses of polymorphic recursion in FGJ and their impact on signatures of generic methods (Section 3.4).

We define a constraint-based algorithm that performs global type inference for FGJ-GT. This algorithm is sound and relatively complete for FGJ programs without polymorphic recursion (Sections 4 and 5).

In Section 7 we show that global type inference is NP-completene.

Our algorithm improves on previous attempts at type inference for Java in the literature as detailed in Section 8.

A prototype of global type inference is available as an evaluated artifact.

A full version of the paper with all proofs is available [28].

2 Motivation

This section presents a sequence of more and more challenging examples for global type inference (GTI). To spice up our examples somewhat, we assume some predefined utility classes with the following interfaces.

```
class Bool {
  Bool not();
}
class Int {
  Int negate ();
  Int add (Int that);
  Int mult (Int that);
}
class Double {
  Double negate ();
  Double add (Double that);
  Double mult (Double that);
}
```

We generally use upper case single-letter identifiers like $\mathtt{X}, \mathtt{Y}, \ldots$ for type variables. Given a FGJ-GT class Cl_0, we call any FGJ class Cl_i that can be transformed to Cl_0 by erasing type annotations a *completion of* Cl_0.

2.1 Multiplication

Here is the FGJ-GT code for multiplying the components of a pair.[5]

```
class MultPair {
  mult (p) { return p.fst.mult(p.snd); }
}
```

Assuming the parameter typing $\mathtt{p} : \mathtt{P}$, result type $\mathtt{R}$, and that `mult` in the body refers to `Int.mult`, we obtain the following constraints.

- From `p.fst`: $\mathtt{P} \lessdot \mathtt{Pair}\langle\mathtt{X}, \mathtt{Y}\rangle$ and $\mathtt{p.fst} : \mathtt{X}$.
- From `p.snd`: $\mathtt{P} \lessdot \mathtt{Pair}\langle\mathtt{Z}, \mathtt{W}\rangle$ and $\mathtt{p.snd} : \mathtt{W}$.
- The two constraints on $\mathtt{P}$ imply that $\mathtt{X} \doteq \mathtt{Z}$ and $\mathtt{Y} \doteq \mathtt{W}$.
- From `.mult (p.snd)`: $\mathtt{X} \lessdot \mathtt{Int}$, $\mathtt{Y} \lessdot \mathtt{Int}$, and $\mathtt{Int} \lessdot \mathtt{R}$.

The return type $\mathtt{R}$ only occurs positively in the constraints, so we can set $\mathtt{R} = \mathtt{Int}$. The argument type $\mathtt{P}$ only occurs negatively in the constraints, so $\mathtt{P} = \mathtt{Pair}\langle\mathtt{X}, \mathtt{Y}\rangle$. This reasoning gives rise to the following completion.

---

[5] We indicate FGJ-GT code fragments by using a gray background.

```
class A1 {
  m(x) { return x.add(x); }
}
class B1 extends A1 {
  m(x) { return x; }
}
```

```
class A2 {
  m(x) { return x; }
}
class B2 extends A2 {
  m(x) { return x.add(x); }
}
```

Figure 3 Method overriding.

```
class MultPair {
  <X extends Int, Y extends Int>
  Int mult (Pair<X,Y> p) { return p.fst.mult(p.snd); }
}
```

We obtain a second completion if we assume that `mult` refers to `Double.mult`.

```
class MultPair {
  <X extends Double, Y extends Double>
  Double mult (Pair<X,Y> p) { return p.fst.mult(p.snd); }
}
```

Finally, the definition of `mult` might be recursive, which generates different constraints for the method invocation of `mult`.

- From `.mult (p.snd)`: $\mathtt{X} \lessdot \mathtt{MultPair}$, $\mathtt{Y} \lessdot \mathtt{P}$, and $\mathtt{R} \lessdot \mathtt{R}$.

Transitivity of subtyping applied to $\mathtt{Y} \lessdot \mathtt{P}$ and $\mathtt{P} \lessdot \mathtt{Pair}\langle\mathtt{X},\mathtt{Y}\rangle$ yields the constraint $\mathtt{Y} \lessdot \mathtt{Pair}\langle\mathtt{X},\mathtt{Y}\rangle$, which triggers the occurs-check in unification and is hence rejected.

The two solutions can be combined to

```
class MultPair {
  <X extends T1, Y extends T2>
  T0 mult (Pair<X,Y> p) { return p.fst.mult(p.snd); }
}
```

where $(T_0, T_1, T_2) \in \{(\mathtt{Int}, \mathtt{Int}, \mathtt{Int}), (\mathtt{Double}, \mathtt{Double}, \mathtt{Double})\}$.

2.2 Inheritance

Let's start with the artificial example in the left listing of Figure 3 and ignore the `Double` class. Type inference proceeds according to the inheritance hierarchy starting from the superclasses. In class `A1`, the inferred method type is `Int A1.m (Int)`. Class `B1` is a subclass of `A1` which must override `m` as there is no overloading in FGJ. However, the inferred method type is `<T> T B1.m(T)`, which is not a correct method override for `A1.m()`. Hence, GTI must instantiate the method type in the subclass `B1` to `Int B1.m(Int)`.

Conversely, for the right listing of Figure 3, GTI infers the types `<T> T A2.m (T)` and `Int B2.m (Int)`. Again, these types do not give rise to a correct method override and GTI is now forced to instantiate the type in the superclass to `Int A2.m (Int)`.

In full Java, type inference would have to offer two alternative results: either two different overloaded methods (one inherited and one local) in `B1`/`B2` or impose the typing `Int B1.m(Int)` or `Int A2.m(Int)` to enforce correct overriding.

```
class Function<S,T> {
  T apply(S arg) { return this.apply (arg); }
}
```

Listing 1 Function class.

2.3 Inheritance and Generics

Suppose we are given a generic class for modeling functions in FGJ (Listing 1). This code is constructed to serve as an “abstract” super class to derive more interesting subclasses. The class `Function<S,T>` must be presented in this explicit way. Its type annotations **cannot** be inferred by GTI because the use of the generic class parameters in the method type cannot be inferred from the implementation.

If we applied GTI to the type-erased version of Listing 1, the `apply` method would be considered a generic method:

```
apply (arg) { ... }
```

-GTI->

```
<A,B> B apply (A arg) { ... }
```

The typing of `apply` in Listing 1 is an instance of this result, so that completeness of GTI is preserved!

Now that we have the abstract class `Function<S,T>` at our disposal, let us apply GTI to a class of boxed values with a `map` function:

```
class Box<S> {
  S val;
  map(f) {
    return new Box<>(f.apply(this.val));
} }
```

GTI finds the following constraints

- the return value must be of type `Box<T>`, for some type `T`,
- `T` is a supertype of the type returned by `f.apply`,
- `apply` is defined in class `Function<S1,T1>` with type `T1 apply(S1 arg)`,
- hence `T1` <: `T` and `S` <: `S1` (because `this.val` : `S`),

and resolves them to the desired outcome where `T1=T` and `S=S1` using the methods of Simonet [26].

```
class Box<S> {
  S val;
  <T> Box<T> map(Function<S,T> f) {
    return new Box<T>(f.apply<S,T>(this.val));
} }
```

But what happens if we add subclasses of `Function`? For example:

```
class Not extends Function<Bool,Bool> {
  apply(b) { return b.not(); }
}
class Negate extends Function<Int,Int> {
  apply(x) { return x.negate(); }
}
```

If we rerun GTI with these classes, we now have additional possibilities to invoke the `apply` method. With `Not`, we need to use the generic type of `Function.apply()`, but instantiate it according to `Function<Bool,Bool>`. Thus, we obtain the constraints `Bool` $\lessdot$ `T` and `S` $\lessdot$ `Bool` for `T = Bool` and `S = Bool`, which are both satisfiable. With `Negate` we run into the same situation with the constraints `Int` $\lessdot$ `Int` and `Int` $\lessdot$ `Int`.

```
class List<A> {
  List<A> add(A item) {...}
  A get() { ... }
}
```

```
class Global{
  m(a){
    return a.add(this).get();
} }
```

Figure 4 Example for multiple inferred types.

Here is another subclass of `Function<S,T>` that we want to consider.

```
class Identity<S> extends Function<S,S> {
  S apply(S arg) { return arg; }
}
```

Here, we obtain the following type constraints

- `apply` is defined in class `Identity<S1>` with type `S1 apply (S1 arg)`,
- hence `S1` $\lessdot$ `T` and `S` $\lessdot$ `S1`.

Resolving the constraints yields `S = T` thus the typing

```
Box<S> map(Identity<S> f);
```

which is an instance of the previous typing.

2.4 Multiple typings

Global type inference processes classes in order of dependency. To see why, consider the classes `List<A>` and `Global` in Figure 4. Class `Global` may depend on class `List` because `Global` uses methods `add` and `get` and `List` defines methods with the same names. The dependency is only approximate because, in general, there may be additional classes providing methods `add` and `get`.

In the example, it is safe to assume that the types for the methods of class `List` are already available, either because they are given (as in the code fragment) or because they were inferred before considering class `Global`.

The method `m` in class `Global` first invokes `add` on `a`, so the type of `a` as well as the return type of `a.add(this)` must be `List<T>`, for some `T`. As `this` has type `Global`, it must be that `Global` is a subtype of `T`, which gives rise to the constraint `Global` $\lessdot$ `T`. By the typing of `get()` we find that the return type of method `m` is also `T`.

But now we are in a dilemma because FGJ only supports *upper bounds* for type variables,[6] so that `Global` $\lessdot$ `T` is not a valid constraint in FGJ. To stay compatible with this restriction, global type inference expands the constraint by instantiating `T` with the (two) superclasses fulfilling the constraint, `Global` and `Object`. They give rise to two incomparable types for `m`, `List<Global> -> Global` and `List<Object> -> Object`. So there are two different FGJ programs that are completions of the `Global` class.

GTI models these instances by inferring an *intersection type* `List<Global> -> Global & List<Object> -> Object` for method `m` and the different FGJ-completions of class `Global` are instances of the intersection type:[7]

[6] Java has the same restriction. Lower bounds are only allowed for wildcards.

[7] The cognoscenti will be reminded of overloading. As FGJ does not support overloading, we rely on resolution by subsequent uses of the method. Moreover, this intersection type cannot be realized by overloading in a Java source program because it is resolved according to the raw classes of the arguments, in this case `List`. It can be realized in bytecode which supports overloading on the return type, too.

```
class UsePair {
  <X,Y> Object prc(Pair<X,Y> p) {
    return this.prc<Y,X>
        (p.swap<X,Y>());
} }
```

```
class UsePair {
  prc(p) {
    return this.prc (p.swap());

} }
```

Figure 5 Example for polymorphic recursion.

```
class Global {
  Global m(List<Global> a) {
    return a.add(this).get();
}
```

```
class Global {
  Object m(List<Object> a) {
    return a.add(this).get();
}
```

In this sense, the inferred intersection type represents a principal typing for the class. Additional classes in the program may further restrict the number of viable types. Suppose we define a class `UseGlobal` as follows:

```
class UseGlobal {
  main() {
    return new Global().m((List<Object>) new List());
} }
```

Due to the dependency on `Global.m()`, type inference considers this class after class `Global`. As it uses `m` at type `List<Object> -> Object`, global type inference narrows the type of `m` to just this alternative.

2.5 Polymorphic recursion

A program uses *polymorphic recursion* if there is a generic method that is invoked recursively at a more specific type than its definition. As a toy example for polymorphic recursion consider the FGJ class `UsePair` with a generic method `prc` that invokes itself recursively on a swapped version of its argument pair (Figure 5, left). This method makes use of polymorphic recursion because the type of the recursive call is different from the declared type of the method. More precisely, the declared argument type is `Pair<X,Y>` whereas the argument of the recursive call has type `Pair<Y,X>` – an instance of the declared type.

For this particular example, global type inference succeeds on the corresponding stripped program shown in Figure 5, right, but it yields a more restrictive typing of `<X> Object prc (Pair<X,X> p)` for the method. A minor variation of the FGJ program with a non-variable instantiation makes type inference fail entirely:

```
class UsePair2 {
  <X,Y> Object prc(Pair<X,Y> p) {
    return this.prc<Y,Pair<X,Y>> (new Pair (p.snd, p));
  }
}
```

Polymorphic recursion is known to make type inference intractable [9, 12] because it can be reduced to an undecidable semi-unification problem [13]. However, *type checking* with polymorphic recursion is tractable and routinely used in languages like Haskell and Java.

GTI does not infer method types with polymorphic recursion. Inference either fails or returns a more restrictive type. Classes making use of polymorphic recursion need to supply explicit typings for methods in question.

$$
\begin{aligned}
&\texttt{T} ::= \texttt{X} \mid \texttt{N}\\
&\texttt{N} ::= \texttt{C<}\overline{\texttt{T}}\texttt{>}\\
&\texttt{L} ::= \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{ N } \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\\
&\texttt{K} ::= \texttt{C}(\overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{f}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\}\\
&\texttt{M} ::= \texttt{m}(\overline{\texttt{x}})\ \{\texttt{return e};\}\\
&\texttt{e} ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid (\texttt{N})\ \texttt{e}
\end{aligned}
$$

Figure 6 Syntax of FGJ-GT.

3 Featherweight Generic Java with Global Type Inference

This section defines the syntax and type system of a modified version of the language Featherweight Generic Java (FGJ) [11], which we call FGJ-GT (with Global Type Inference). The main omissions with respect to FGJ are method types specifications and polymorphic recursion. We finish the section by formally connecting FGJ and FGJ-GT and by establishing some properties about polymorphic recursion in FGJ.

3.1 Syntax

Figure 6 defines the syntax of FGJ-GT. Compared to FGJ, type annotations for method parameters and method return types are omitted. Object creation via `new` as well as method calls come do not require instantiation of their generic parameters. We keep the class constraints $\overline{\texttt{X}} \lhd \overline{\texttt{N}}$ as well as the types for fields $\overline{\texttt{T}}\ \overline{\texttt{f}}$ as we consider them as part of the specification of a class.

We make the following assumptions for the input program:

- All types `N` and `T` are well formed according to the rules of FGJ, which carry over to FGJ-GT (see Fig. 8).
- The methods of a class call each other mutually recursively.
- The classes in the input are topologically sorted so that later classes only call methods in classes that come earlier in the sorting order.

Our requirements on the method calls do not impose serious restrictions as any class, say `C`, can be transformed to meet them as follows. A preliminary dependency analysis determines an approximate call graph. We cluster the methods of `C` according to the n strongly connected components of the call graph. Then we split the class into a class hierarchy $\texttt{C}_1 \lhd \ldots \lhd \texttt{C}_n$ such that each class $\texttt{C}_i$ contains exactly the methods of one strongly connected component and assign a method cluster to $\texttt{C}_i$ if all calls to methods of `C` now target methods assigned to $\texttt{C}_j$, for some $j \geq i$. The class $\texttt{C}_1$ replaces `C` everywhere in the program: in subtype bounds, in `new` expressions, and in casts. More precisely, if `C` is defined by $\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ldots$, then the class headers for the $\texttt{C}_i$ are defined as follows:

- $\texttt{class C}_\texttt{i}\texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{C}_{\texttt{i+1}}\texttt{<}\overline{\texttt{X}}\texttt{>}\ldots$, for $1 \leq i < n$ and
- $\texttt{class C}_\texttt{n}\texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ldots$.

It follows from this discussion that the resulting classes have to be processed backwards starting with $\texttt{C}_n, \texttt{C}_{n-1}, \ldots, \texttt{C}_1$. Figure 7 showcases this process with a short example.

```
class C extends Object {
  m1(a){
    return a;
  }
  m2(b){
    return this.id(a);
  }
}
```

(a) The methods `m1` and `m2` can be separated.

```
class C1 extends C2 {
  m2(b){
    return this.id(a);
  }
}
class C2 extends Object {
  m1(a){
    return a;
  }
}
```

(b) After the transformation.

Figure 7 Example for splitting a class into its strongly connected components.

3.2 Typing

We start with some notation. An environment Γ is a finite mapping from variables to types, written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$; a type environment Δ is a finite mapping from type variables to nonvariable types, written $\overline{\mathtt{X}} <: \overline{\mathtt{N}}$, which takes each type variable to its bound. As in FGJ, we do not impose an ordering on environment entries to enable F-bounded polymorphism.

There is a new method environment Π which maps pairs of a class header $\mathtt{C}<\overline{\mathtt{X}}>$ and a method name `m` to a set of method types of the form $<\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}>\overline{\mathtt{T}} \to \mathtt{T}$. It supports the *mtype* function that relates a nonvariable type `N` and a method name `m` to a method type.

The judgments for subtyping $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and well-formedness of types $\Delta \vdash \mathtt{T}\ \mathtt{ok}$ (Figure 8) stay the same as in FGJ.

The overall approach to typing changes with respect to FGJ. In FGJ, classes can be checked in any order as the method typings of all other classes are available in the syntax. FGJ-GT processes classes in order such that early classes do not invoke methods in late classes.

The new program typing rule (GT-PROGRAM) for the judgment $\vdash \overline{\mathtt{L}} : \Pi$ reflects this approach. It starts with an empty method environment and applies class typing to each class in the sequence provided. Each processed class adds its method typings to the method environment which is threaded through to constitute the program type as the final method environment Π.

Expression typing $\Pi; \Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ changes subtly (see Figure 9). As FGJ-GT omits some type annotations, we are forced to adapt some of FGJ's typing rules. The new rules infer omitted types and disable polymorphic recursion.

The new method environment Π is only used in the revised rule for method invocation (GT-INVK), where it is passed as an additional parameter to *mtype*. The revised definition of *mtype* (Figure 10) locates the class that contains the method definition by traversing the subtype hierarchy and looks up the method type in environment Π, which contains the method types that were already inferred. Our definition of *mtype* does not support overloading as Π relate at most one type to each method definition (cf. rule (GR-CLASS)). The instantiation of the method's type parameters is inferred in FGJ-GT.

The rule (GT-NEW) changes to infer the instantiation of the class's type parameters: the rule simply assumes a suitable instantiation by some $\overline{\mathtt{U}}$.

Finally, (GT-CAST) replaces the three rules (GT-UCAST'), (GT-DCAST'), and (GT-SCAST') of FGJ. This is a slight simplification with respect to FGJ. While the three original rules cover disjoint use cases (upcast, downcast, and stupid cast that is sure to fail) of the cast operation, they are not exhaustive! The rule (GT-DCAST') only admits downcasts that

Subtyping:

$$\Delta \vdash \texttt{T} <: \texttt{T} \qquad \text{(S-REFL)}$$

$$\frac{\Delta \vdash \texttt{S} <: \texttt{T} \qquad \Delta \vdash \texttt{T} <: \texttt{U}}{\Delta \vdash \texttt{S} <: \texttt{U}} \qquad \text{(S-TRANS)}$$

$$\Delta \vdash \texttt{X} <: \Delta(\texttt{X}) \qquad \text{(S-VAR)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N}\{\ldots\}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}} \qquad \text{(S-CLASS)}$$

Well-formed types:

$$\Delta \vdash \texttt{Object ok} \qquad \text{(WF-OBJECT)}$$

$$\frac{\texttt{X} \in dom(\Delta)}{\Delta \vdash \texttt{X ok}} \qquad \text{(WF-VAR)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N}\{\ldots\} \quad \Delta \vdash \overline{\texttt{T}}\ \texttt{ok} \qquad \Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{> ok}} \qquad \text{(WF-CLASS)}$$

Figure 8 Well-formedness and subtyping.

work the same in a type-passing semantics as in a type erasure semantics. We elide this distinction for simplicity, though it could be handled by introducing constraints analogous to the *dcast* function from FGJ.

The typing rule for a method `m`, (GT-METHOD), changes significantly. By our assumption on the order, in which classes are processed, the typing of `m` is already provided by the method environment Π. The type environment Δ is also provided as an input. Moreover, to rule out polymorphic recursion, the assumptions about the local methods of class `C` are monomorphic at this stage. The rule type checks the body for the inferred type of method `m`.

All this information is provided and generated by the rule for class typing, (GT-CLASS). A class typing for `C` receives an incoming method type environment Π and generates an extended one Π'' which additionally contains the method types inferred for `C`.

In Π', we generate some monomorphic types for all methods of class `C`. We use these types to check the methods. Afterwards, we return generalized versions of these same types in Π''. All method types use the same generic type variables $\overline{\texttt{Y}}$ with the same constraints $\overline{\texttt{P}}$. It is safe to make this assumption in the absence of polymorphic recursion as we will show in Proposition 5.

3.3 Soundness of Typing

We show that every typing derived by the FGJ-GT rules gives rise to a completion, that is, a well-typed FGJ program with the same structure.

▶ **Definition 1** (Erasure). *Let* $\texttt{e}'$, $\texttt{M}'$, $\texttt{K}'$, $\texttt{L}'$ *be expression, method definition, constructor definition, class definition for FGJ. Define erasure functions* $|\texttt{e}'|$, $|\texttt{M}'|$, $|\texttt{K}'|$, $|\texttt{L}'|$ *that map to the corresponding syntactic categories of FGJ-GT as shown in Figure 11.*

▶ **Definition 2** (Completion). *An FGJ expression* $\texttt{e}'$ *is a* completion *of a FGJ-GT expression* $\texttt{e}$ *if* $\texttt{e} = |\texttt{e}'|$. *Completions for method definitions, constructor definitions, and class definitions are defined analogously.*

Expression typing:

$$\Pi; \Delta; \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(GT-VAR)}$$

$$\frac{\Pi; \Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Pi; \Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}_i} \qquad \text{(GT-FIELD)}$$

$$\frac{\begin{array}{c}\Pi; \Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0), \Pi) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}}\ \mathtt{ok} \qquad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \qquad \Pi; \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\end{array}}{\Pi; \Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}} \qquad \text{(GT-INVK)}$$

$$\frac{\Delta \vdash \mathtt{N}\ \mathtt{ok} \quad \mathtt{N} = \mathtt{C}\mathtt{<}\overline{\mathtt{U}}\mathtt{>} \quad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \Pi; \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \quad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}}{\Pi; \Delta; \Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{N}} \qquad \text{(GT-NEW)}$$

$$\frac{\Pi; \Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0}{\Pi; \Delta; \Gamma \vdash (\mathtt{N})\mathtt{e}_0 : \mathtt{N}} \qquad \text{(GT-CAST)}$$

Method typing:

$$\frac{\begin{array}{c}\forall \overline{\mathtt{T}}, \mathtt{T} : \mathtt{<>}\overline{\mathtt{T}} \to \mathtt{T} \in \Pi(\mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}.\mathtt{m}) \qquad \Delta \vdash \mathtt{S} <: \mathtt{T} \\ \Pi; \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}},\ \mathtt{this} : \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 : \mathtt{S} \\ \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}, \Pi)\end{array}}{\Pi, \Delta \vdash \mathtt{m}(\overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e}_0;\}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \mathtt{with}\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}} \qquad \text{(GT-METHOD)}$$

Class typing:

$$\frac{\begin{array}{c}\Pi' = \Pi \cup \{\mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}.\mathtt{m} \mapsto \mathtt{<>}\overline{\mathtt{T}_m} \to \mathtt{T}_m \mid \mathtt{m} \in \overline{\mathtt{M}}\} \\ \Pi'' = \Pi \cup \{\mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}.\mathtt{m} \mapsto \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}_m} \to \mathtt{T}_m \mid \mathtt{m} \in \overline{\mathtt{M}}\} \\ \Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{P}}\ \mathtt{ok} \qquad \forall \mathtt{m} : \Delta \vdash \overline{\mathtt{T}_m}, \mathtt{T}_m\ \mathtt{ok} \\ \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{N}, \overline{\mathtt{T}}\ \mathtt{ok} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \\ \Pi', \Delta \vdash \overline{\mathtt{M}}\ \mathtt{OK}\ \mathtt{IN}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \mathtt{with}\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>} \\ \mathtt{K} = \mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}\end{array}}{\Pi \vdash \mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK} : \Pi''} \qquad \text{(GT-CLASS)}$$

Program typing:

$$\frac{\emptyset \vdash \mathtt{L}_1 : \Pi_1 \quad \Pi_1 \vdash \mathtt{L}_2 : \Pi_2 \quad \ldots \quad \Pi_{n-1} \vdash \mathtt{L}_n : \Pi_n}{\vdash \overline{\mathtt{L}} : \Pi_n} \qquad \text{(GT-PROGRAM)}$$

Figure 9 Typing rules.

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \text{(F-OBJECT)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N } \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \mathit{fields}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{g}}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}} \qquad \text{(F-CLASS)}$$

Method type lookup:

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \in \overline{\texttt{M}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \to \texttt{U} \in \Pi(\texttt{C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>.m})\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}, \Pi) = [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \to \texttt{U}} \qquad \text{(MT-CLASS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}, \Pi) = \mathit{mtype}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}, \Pi)} \qquad \text{(MT-SUPER)}$$

Valid method overriding:

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{N}, \Pi) = \texttt{<}\overline{\texttt{Z}} \triangleleft \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}} \to \texttt{U} \text{ implies } \overline{\texttt{P}}, \overline{\texttt{T}} = [\overline{\texttt{Y}}/\overline{\texttt{Z}}](\overline{\texttt{Q}}, \overline{\texttt{U}}) \text{ and } \overline{\texttt{Y}} \texttt{ <: } \overline{\texttt{P}} \vdash \texttt{T}_0 \texttt{ <: } [\overline{\texttt{Y}}/\overline{\texttt{Z}}]\texttt{U}_0}{\mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}_0, \Pi)}$$

Figure 10 Auxiliary functions.

$$\begin{aligned}
|\texttt{x}| &= \texttt{x} \\
|\texttt{e.f}| &= |\texttt{e}|\texttt{.f} \\
|\texttt{e<}\overline{\texttt{T}}\texttt{>.m}(\overline{\texttt{e}})| &= |\texttt{e}|\texttt{.m}(|\overline{\texttt{e}}|) \\
|\texttt{new C<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{e}})| &= \texttt{new C}(|\overline{\texttt{e}}|) \\
|(\texttt{N})\ \texttt{e}| &= (\texttt{N})\ |\texttt{e}| \\
|\texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\texttt{return e};\}| &= \texttt{m}(\overline{\texttt{x}})\ \{\texttt{return } |\texttt{e}|;\} \\
|\texttt{C}(\overline{\texttt{U}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{g}});\ \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\}| &= \texttt{C}(\overline{\texttt{g}}, \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{g}});\ \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\} \\
|\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{ N } \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}| &= \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{ N } \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ |\texttt{K}|\ |\overline{\texttt{M}}|\}
\end{aligned}$$

Figure 11 Erasure functions.

▶ **Theorem 3.** *Suppose that* $\vdash \overline{\texttt{L}} : \Pi$ *such that* $|\Pi(\texttt{C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>.m})| = 1$*, for all* C.m *defined in* $\overline{\texttt{L}}$*. Then there is a completion* $\overline{\texttt{L}}'$ *of* $\overline{\texttt{L}}$ *such that* $\overline{\texttt{L}}'$ OK *is derivable in FGJ.*

Proof. The proof is by induction on the length of $\overline{\texttt{L}}$.

Consider the class typing $\Pi \vdash \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK} : \Pi''$ for an element of $\overline{\texttt{L}}$.

We assume that all classes before L are completed according to the incoming Π: If $\Pi(\texttt{D<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>.n}) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}$, then $\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\ \texttt{T}\ \texttt{n}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ldots$ is in the completion of D.

Clearly, we can construct a completion for the class, if we can do so for each method. So we have to construct $\overline{\texttt{M}}'$ such that $\overline{\texttt{M}}'\ \texttt{OK IN C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>}$.

Inversion of (GT-CLASS) yields

$$\Pi' = \Pi \cup \{\texttt{C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>.m} \mapsto \texttt{<>}\overline{\texttt{T}_\texttt{m}} \to \texttt{T}_\texttt{m} \mid \texttt{m} \in \overline{\texttt{M}}\} \quad (1)$$

$$\Pi'' = \Pi \cup \{\texttt{C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>.m} \mapsto \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}_\texttt{m}} \to \texttt{T}_\texttt{m} \mid \texttt{m} \in \overline{\texttt{M}}\} \quad (2)$$

$$\Pi', \Delta \vdash \overline{\texttt{M}}\ \texttt{OK IN C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N with <}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>} \quad (3)$$

$$\Delta = \overline{\texttt{X}} \texttt{ <: } \overline{\texttt{N}}, \overline{\texttt{Y}} \texttt{ <: } \overline{\texttt{P}} \quad (4)$$

Given some $\texttt{M} = \texttt{m}(\overline{\texttt{x}})\{\texttt{return e}_0;\} \in \overline{\texttt{M}}$, we show that

$$\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\ \texttt{T}_\texttt{m}\ \texttt{m}(\overline{\texttt{T}_\texttt{m}}\ \overline{\texttt{x}})\{\texttt{return e}'_0;\}\ \texttt{OK IN C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \quad (5)$$

is derivable for such completion $\texttt{e}'_0$ of $\texttt{e}_0$.

By inversion of (3) for M, we obtain

$$\mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}_\texttt{m}} \to \texttt{T}_\texttt{m}, \Pi) \quad (6)$$

$$\Pi; \Delta; \overline{\texttt{x}} : \overline{\texttt{T}_\texttt{m}},\ \texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>} \vdash \texttt{e}_0 : \texttt{S} \quad (7)$$

$$\Delta \vdash \texttt{S <: T}_\texttt{m} \quad (8)$$

As Δ in (4) is defined as in (GT-METHOD'), the well-formedness judgments are all given, the subtyping judgment (8) is given as well as the override (7), the rule (GT-METHOD') applies if we can establish

$$\Delta; \overline{\texttt{x}} : \overline{\texttt{T}_\texttt{m}},\ \texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>} \vdash \texttt{e}'_0 : \texttt{S} \quad (9)$$

for a completion of $\texttt{e}_0$.

To see that, we need to consider the rules (GT-NEW), (GT-CAST), and (GT-INVK). The (GT-NEW) rule poses the existence of some $\overline{\texttt{U}}$ such that $\texttt{N} = \texttt{C<}\overline{\texttt{U}}\texttt{>}$ for checking $\texttt{e} = \texttt{new C}(\overline{\texttt{e}}) : \texttt{N}$. In the completion, we define $\texttt{e}' = \texttt{new N}(\overline{\texttt{e}}') : \texttt{N}$ to apply rule (GT-NEW') to the completions of the arguments.

The rule (GT-CAST) splits into three rules (GT-UCAST'), (GT-DCAST'), and (GT-SCAST'). These rules are disjoint, so that at most one of them applies to each occurrence of a cast. Here we assume a more liberal version of (GT-DCAST') that admits downcasts that are not stable under type erasure semantics.

For the rule (GT-INVK), we first consider calls to methods not defined in the current class. By our assumption on previously checked classes D and their methods n, $\mathit{mtype}(\texttt{n}, \texttt{D}, \Pi) = \{\mathit{mtype}'(\texttt{n}, \texttt{D}')\}$ where the right side lookup happens in the completion following the definitions for FGJ (i.e., $\texttt{D}'$ is the completion for D). The (GT-INVK) rule poses the existence of some $\overline{\texttt{V}}$ that satisfies the same conditions as in (GT-INVK'). Hence, we define the completion of $\texttt{e}_0.\texttt{n}(\overline{\texttt{e}}) : [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}$ as $\texttt{e}'_0.\texttt{n<}\overline{\texttt{V}}\texttt{>}(\overline{\texttt{e}}') : [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}$.

Next we consider calls to methods $\mathtt{n}$ defined in the current class, say, $\mathtt{C}$. For those methods, $\mathit{mtype}(\mathtt{n}, \mathtt{C}, \Pi) = \mathtt{<>}\overline{\mathtt{U}} \to \mathtt{U}$, a non-generic type. By the definition of Π'', we know that the type of this method will be published in the completion as $\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}$. Hence, $\mathit{mtype}'(\mathtt{n}, \mathtt{C}') = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}$. As methods in $\mathtt{C}$ are mutually recursive, the rule must pose that $\overline{\mathtt{V}} = \overline{\mathtt{Y}}$ (cf. Proposition 5). This setting fulfills all assumptions:

$$\Delta \vdash \overline{\mathtt{Y}}\ \mathtt{ok} \tag{10}$$

$$\Delta \vdash \overline{\mathtt{Y}} \mathtt{<:} [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \tag{11}$$

We set the completion of $\mathtt{e}_0.\mathtt{n}(\overline{\mathtt{e}}) : [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}]\mathtt{U}$ to $\mathtt{e}'_0.\mathtt{n}\mathtt{<}\overline{\mathtt{Y}}\mathtt{>}(\overline{\mathtt{e}}') : [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}]\mathtt{U}$, which is derivable in FGJ.

The remaining expression typing rules are shared between FGJ and FGJ-GT, so they do not affect completions. ◀

3.4 Polymorphic Recursion, Formally

Consider an FGJ class $\mathtt{C}$ with n mutually recursive methods $\mathtt{m}_i : \forall \overline{\mathtt{X}}_i.\overline{\mathtt{A}}_i \to \overline{\mathtt{A}}_i$, for $1 \le i \le n$. Define the *instantiation multigraph* $IG(\mathtt{C})$ as a directed multigraph with vertices $\{1, \dots, n\}$. Edges between i and j in this graph are labeled with a substitution from $\overline{\mathtt{X}}_j$ to types in $\mathtt{m}_i$, which may contain type variables from $\overline{\mathtt{X}}_i$. In particular, if $\mathtt{m}_i$ invokes $\mathtt{m}_j$ where the generic type variables in the type of $\mathtt{m}_j$ are instantiated with substitution $\overline{\mathtt{U}}/\overline{\mathtt{X}}_j$ (see rule GT-INVK), then $i \xrightarrow{\overline{\mathtt{U}}/\overline{\mathtt{X}}_j} j$ is an edge of $IG(\mathtt{C})$.

Define the *closure of the instantiation multigraph* $IG^*(\mathtt{C})$ as the multigraph obtained from $IG(\mathtt{C})$ by applying the following rule, which composes the instantiating substitutions, exhaustively:

$$i \xrightarrow{\overline{\mathtt{U}}/\overline{\mathtt{X}}_j} j \quad \wedge \quad j \xrightarrow{\overline{\mathtt{V}}/\overline{\mathtt{X}}_k} k \qquad \Rightarrow \qquad i \xrightarrow{[\overline{\mathtt{U}}/\overline{\mathtt{X}}_j]\overline{\mathtt{V}}/\overline{\mathtt{X}}_k} k \tag{12}$$

▶ **Definition 4.** *Method* $\mathtt{m}_i$ *is* involved in polymorphic recursion *if there is an edge*

$$i \xrightarrow{\overline{\mathtt{W}}/\overline{\mathtt{X}}_i} i \quad \in IG^*(\mathtt{C}) \qquad \textit{such that} \qquad \overline{\mathtt{W}} \neq \overline{\mathtt{X}}_i \tag{13}$$

For the toy example in Figure 5, we obtain the multigraph $IG^*(\mathtt{UsePair})$ which indicates that `prc` is involved in polymorphic recursion:

$IG(\mathtt{UsePair})$	$IG^*(\mathtt{UsePair})$	
$\mathtt{prc} \xrightarrow{\mathtt{Y,X/X,Y}} \mathtt{prc}$	$\mathtt{prc} \xrightarrow{\mathtt{Y,X/X,Y}} \mathtt{prc}$	$\mathtt{prc} \xrightarrow{\mathtt{X,Y/X,Y}} \mathtt{prc}$

The call to `swap` does not appear in the graph because `swap` is defined in a different class.

For `UsePair2`, we obtain a multigraph $IG^*(\mathtt{UsePair2})$ with infinitely many edges which is also clear indication for polymorphic recursion:

$IG(\mathtt{UsePair2})$	$IG^*(\mathtt{UsePair2})$
$\mathtt{prc} \xrightarrow{\mathtt{Y,Pair<X,Y>/XY}} \mathtt{prc}$	$\mathtt{prc} \xrightarrow{\mathtt{Y,Pair<X,Y>/XY}} \mathtt{prc}$
	$\mathtt{prc} \xrightarrow{\mathtt{Pair<X,Y>,Pair<Y,Pair<X,Y>>/XY}} \mathtt{prc}$
	...

Clearly, $IG(\mathtt{C})$ is finite and can be constructed effectively by collecting the instantiating substitutions from all method call sites. Repeated application of the propagation rule (12) either results in saturation where no edge of the resulting multigraph satisfies (13) or it detects an instantiating edge as in condition (13).

The following condition is necessary for the absence of polymorphic recursion.

▶ **Proposition 5.** *Suppose an FGJ class* C *has n methods, which are mutually recursive. If* C *does not exhibit polymorphic recursion, then*

- *all methods quantify over the same number of generic variables;*
- *if a method has generic variables* $\overline{\mathtt{X}}$*, then each call to a method of* C *instantiates with a permutation of the* $\overline{\mathtt{X}}$*;*
- $IG^*(\mathtt{C})$ *is finite.*

Proof. Suppose for a contradiction that there are two distinct methods $\mathtt{m}_i$ and $\mathtt{m}_j$ with generic variables $\overline{\mathtt{X}}_i$ and $\overline{\mathtt{X}}_j$, respectively, where $|\overline{\mathtt{X}}_i| < |\overline{\mathtt{X}}_j|$. By mutual recursion, $\mathtt{m}_i$ invokes $\mathtt{m}_j$ directly or indirectly and vice versa. Hence, $IG^*(\mathtt{C})$ contains edges from i to j and back:

$$i \xrightarrow{\overline{\mathtt{U}}/\overline{\mathtt{X}}_j} j \qquad j \xrightarrow{\overline{\mathtt{V}}/\overline{\mathtt{X}}_i} i$$

As $IG^*(\mathtt{C})$ is closed under composition, it must also contain the edge

$$j \xrightarrow{[\overline{\mathtt{V}}/\overline{\mathtt{X}}_i]\overline{\mathtt{U}}/\overline{\mathtt{X}}_j} j.$$

By assumption C does not use polymorphic recursion, so it must be that $[\overline{\mathtt{V}}/\overline{\mathtt{X}}_i]\overline{\mathtt{U}}/\overline{\mathtt{X}}_j = \overline{\mathtt{X}}_j/\overline{\mathtt{X}}_j$. To fulfill this condition, all components of $\overline{\mathtt{U}}$ must be variables $\in \overline{\mathtt{X}}_i$. As $|\overline{\mathtt{X}}_i| < |\overline{\mathtt{X}}_j| = |\overline{\mathtt{U}}|$, there must be some variable $\mathtt{X} \in \overline{\mathtt{X}}_i$ that occurs more than once in $\overline{\mathtt{U}}$, say, at positions j_1 and j_2. But that means the variables at positions j_1 and j_2 in $\overline{\mathtt{X}}_j$ are mapped to the same component of $\overline{\mathtt{V}}$. This is a contradiction because this substitution cannot be the identity substitution $\overline{\mathtt{X}}_j/\overline{\mathtt{X}}_j$.

Hence, all methods have the same number of generic variables and all instantiations must use variables.

Suppose now that there is a direct call from $\mathtt{m_i}$ to $\mathtt{m}_j$ where the instantiation $\overline{\mathtt{U}}/\overline{\mathtt{X}}_j$ is not a permutation. Hence, there is a variable that appears more than once in $\overline{\mathtt{U}}$, which leads to a contradiction using similar reasoning as before.

Hence, all instantiations must be permutations over a finite set of variables, so that $IG^*(\mathtt{C})$ is finite. ◀

Moreover, if a class has only mutually recursive methods without polymorphic recursion, we can assume that each method uses the same generic variables, say $\overline{\mathtt{X}}$, and each instantiation for class-internal method calls is the identity $\overline{\mathtt{X}}/\overline{\mathtt{X}}$.

Using the same generic variables is achieved by α conversion. By Proposition 5, we already know that each instantiation is a permutation. Each self-recursive call must use an identity instantiation already, otherwise it would constitute an instance of polymorphic recursion. Suppose that method m calls method n instantiated with a non-identity permutation, say π so that parameter $\mathtt{X_i}$ of n gets instantiated with $\mathtt{X}_{\pi(\mathtt{i})}$ of m. In this case, we reorder the generic parameters of n according to the inverse permutation π^{-1} and propagate this permutation to all call sites of n. For the call in m, we obtain the identity permutation $\pi \cdot \pi^{-1}$, for self-recursive calls inside n, the instantiation remains the identity (for the same reason), for a call site in another method which instantiates n with permutation σ, we change that permutation to $\sigma \cdot \pi^{-1}$, which is again a permutation. This way, we can eliminate all non-identity instantiations from calls inside m.

We move our attention to n. Each self-recursive call and each call to m uses the identity instantiation, the latter by construction. So we only need to consider calls to $\mathtt{n}' \notin \{\mathtt{n}, \mathtt{m}\}$ with an instantiation which is not the identity permutation. We can also assume that $\mathtt{n}'$ is not called from m: otherwise, n' would have the generic variables in the same order as m and hence as n. But that means we can fix all calls to $\mathtt{n}'$ by applying the inverse permutations as for n *without disturbing the already established identity instantiations.*

```
class C1 extends Object {
  m1(){ return new C2().m2(); }
}
class C2 extends Object{
  m2(){ return new C1().m1(); }
}
```

Figure 12 Invalid FGJ-GT program.

```
class D1 extends Object {
  m(){ return ...; }
}
class D2 extends Object{
  m1(x){ return new D2().m2(); }
  m2( ){ return new D2().m1(
                 new D1().m()); }
}
```

Figure 13 Valid FGJ-GT program.

Each such step eliminates all non-identity instantiations for at least one method without disturbing previous identity instantiations. Hence, the procedure terminates after finitely many steps with a class with all instantiations being identity permutations.

4 Type inference algorithm

This section presents our type inference algorithm. The algorithm is given method assumptions Π and applied to a single class L at a time:

$$\begin{aligned}
&\textbf{FJTypeInference}(\Pi, \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\ldots\}) = \\
&\qquad \textbf{let}\ (\overline{\lambda}, C) \quad = \textbf{FJType}(\Pi, \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\ldots\}) \quad \text{// constraint generation} \\
&\qquad\quad (\sigma, \overline{\texttt{Y}} \lhd \overline{\texttt{P}}) \quad = \textbf{Unify}(C, \overline{\texttt{X}} <: \overline{\texttt{N}}) \qquad\qquad \text{// constraint solving} \\
&\textbf{in}\ \Pi \cup \{(\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m} : \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \overline{\sigma(a)} \to \sigma(a)) \mid (\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m} : \overline{a} \to a) \in \overline{\lambda}\}
\end{aligned}$$

The overall algorithm is nondeterministic. The function **Unify** may return finitely many times as there may be multiple solutions for a constraint set. A local solution for class C may not be compatible with the constraints generated for a subsequent class. In this case, we have to backtrack to C and proceed to the next local solution; if thats fail we have to backtrack further to an earlier class.

4.1 Type inference for a program

Type inference processes a program one class at a time. To do so, it must be possible to order the classes such that early classes never call methods in later classes. As an example, Figure 12 shows a program that is acceptable in FGJ, but rejected by FGJ-GT because the methods m1 and m2 are mutually recursive across class boundaries. There is no order in which classes C1 and C2 can be processed.

Figure 13 contains a program acceptable to both FGJ-GT and FGJ because the mutual recursion of methods m1 and m2 is taking place inside class D2. As D2 invokes method m of D1, type inference must process D1 before D2, which corresponds to the constraints imposed by the typing of FGJ-GT in Section 3.2.

We obtain a viable order for processing the class declarations by computing an approximate call graph based solely on method names. That is, if method m is used in C3 and defined both in C1 and C2, then C1 and C2 must both be processed before C3. In such a case, the use of m might be ambiguous so that type inference for class C3 proposes more than one solution. Global type inference attempts to extend each partial solution to a solution for the whole program and backtracks if that fails.

$T, U ::= a \mid \mathtt{X} \mid N$	type variable, bounded type parameter, or type
$N ::= \mathtt{C}\texttt{<}\overline{T}\texttt{>}$	class type (with type variables)
$sc ::= T \lessdot U \mid T \doteq U$	simple constraint: subtype or equality
$oc ::= \{\{\overline{sc_1}\}, \ldots, \{\overline{sc_n}\}\}$	or-constraint
$c ::= sc \mid oc$	constraint
$C ::= \{\overline{c}\}$	constraint set
$\lambda ::= \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}.\mathtt{m} : \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\ \overline{T} \to T$	method type assumption
$\eta ::= \mathtt{x} : T$	parameter assumption
$\Pi ::= \Pi \cup \overline{\lambda}$	method type environment
$\Theta ::= (\Pi; \overline{\eta})$	

Figure 14 Syntax of constraints and type assumptions.

$$
\begin{array}{l}
\textbf{FJType}(\Pi, \texttt{class C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{> extends N}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\,\overline{\mathtt{M}}\}) = \\
\quad \textbf{let} \quad \overline{a}_{\mathtt{m}} \text{ be fresh type variables for each } \mathtt{m} \in \overline{\mathtt{M}} \\
\qquad \overline{\lambda}_o = \{\mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}.\mathtt{m} : \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\ \overline{\mathtt{T}} \to a_{\mathtt{m}} \mid \mathtt{m} \in \overline{\mathtt{M}}, \mathit{mtype}(\mathtt{m}, \mathtt{N}, \Pi) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{T}} \to \mathtt{T}\} \\
\qquad C_o = \{a_{\mathtt{m}} \lessdot \mathtt{T} \mid \mathtt{m} \in \overline{\mathtt{M}}, \mathit{mtype}(\mathtt{m}, \mathtt{N}, \Pi) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{T}} \to \mathtt{T}\} \\
\qquad \overline{\lambda}' = \{(\mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}.\mathtt{m} : \overline{a} \to a_{\mathtt{m}}) \mid \mathtt{m} \in \overline{\mathtt{M}}, \mathit{mtype}(\mathtt{m}, \mathtt{N}, \Pi) \text{ not defined}, \overline{a} \text{ fresh}\} \\
\qquad C_m = \{\{a_{\mathtt{m}} \lessdot \mathtt{Object}, \overline{a} \lessdot \overline{\mathtt{Object}}\} \mid (\mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}.\mathtt{m} : \overline{a} \to a_{\mathtt{m}}) \in \overline{\lambda}'\} \\
\qquad \Pi = \Pi \cup \overline{\lambda}' \cup \overline{\lambda}_o \\
\quad \textbf{in} \quad (\Pi, C_o \cup C_m \cup \bigcup_{\mathtt{m} \in \overline{\mathtt{M}}} \textbf{TYPEMethod}(\Pi, \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}, \mathtt{m}))
\end{array}
$$

Figure 15 Constraint generation for classes.

4.2 Constraint generation

Figure 14 defines the syntax of constraints. We extend types with *type variables* ranged over by a. A constraint is either a simple constraint sc or an or-constraint oc, which is a set of sets of simple constraints. An or-constraint represents different alternatives, similar to an intersection type, and cannot be nested. The output of constraint generation is a set of constraints C, which can hold simple constraints as well as or-constraints.

Figure 15 contains the algorithm **FJType** to generate constraints for classes. Its input consists of the method type environment Π of the previously checked classes. It distinguishes between overriding and non-overriding method definitions. The former are recognized by successful lookup of their type using *mtype*. We set up the method type assumptions accordingly and generate a constraint between the inferred return type $a_{\mathtt{m}}$ and the one of the overridden method to allow for covariant overriding. Constraints for the latter methods are generated with all fresh type variables for the argument and result types.

Constraint generation alternates with constraint solving: After generating constraints with **FJType**, we solve them to obtain one or more candidate extensions for the method type environment Π. Next, we pick a candidate and continue with the next class until all classes are checked and we have an overall method type environment. Otherwise, we backtrack to check the next candidate.

$$\begin{array}{ll}\textbf{TYPEMethod}(\Pi, \texttt{C<}\overline{\texttt{X}}\texttt{>}, \texttt{m}(\overline{\texttt{x}})\{\texttt{return e};\}) = & \\ \quad\textbf{let} & \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\ \overline{T} \to T = \Pi(\texttt{C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>.m}) \\ & (R, C) = \textbf{TYPEExpr}(\Pi; \{\texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>}\} \cup \{\overline{\texttt{x}} : \overline{T}\}), \texttt{e}) \\ \quad\textbf{in} & C \cup \{R \lessdot T\}\end{array}$$

The **TYPEMethod** function for methods calls the **TYPEExpr** function with the return expression. It adds the assumptions for `this` and for the method parameters to the global assumptions before passing them to **TYPEExpr**.

In the following we define the **TYPEExpr** function for every possible expression: $\textbf{TYPEExpr} : \Theta \times \texttt{Expression} \to T \times C$

$$\textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{x}) = (\overline{\eta}(\texttt{x}), \emptyset)$$

When we encounter a field `e.f`, we consider all classes `C` that define field `f` and impose an or-constraint that covers all alternatives: the type R of the expression `e` must be a subtype of a generic instance of `C` and the return type must be the corresponding field type.

$$\begin{array}{ll}\textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e.f}) = & \\ \quad\textbf{let} & (R, C_R) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}) \\ & a \text{ fresh} \\ & c = oc\{\{R \lessdot \texttt{C<}\overline{a}\texttt{>}, a \doteq [\overline{a}/\overline{\texttt{X}}]\texttt{T}, \overline{a} \lessdot [\overline{a}/\overline{\texttt{X}}]\overline{\texttt{N}} \mid \overline{a} \text{ fresh}\} \\ & \qquad\qquad \mid \texttt{T f} \in \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>}\{\overline{\texttt{T}}\ \overline{\texttt{f}}; \ldots\}\} \\ \quad\textbf{in} & (a, (C_R \cup \{c\}))\end{array}$$

We treat method calls in a similar way. We impose an or-constraint that considers a generic instance of a method type in a class providing that method (with the same number of parameters). Each choice imposes a subtyping constraint on the receiver type R as well as subtyping constraints on the argument types $\overline{R}$. Moreover, we need to check that the subtyping constraints of the method type are obeyed by instantiating them accordingly.

$$\begin{array}{ll}\textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e.m}(\overline{\texttt{e}})) = & \\ \quad\textbf{let} & (R, C_R) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}) \\ & \forall \texttt{e}_i \in \overline{\texttt{e}} : (R_i, C_i) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}_i) \\ & a \text{ fresh} \\ & c = oc\{\{\ R \lessdot \texttt{C<}\overline{a}\texttt{>}, a \doteq [\overline{b}/\overline{\texttt{Y}}][\overline{a}/\overline{\texttt{X}}]T, \overline{R} \lessdot [\overline{b}/\overline{\texttt{Y}}][\overline{a}/\overline{\texttt{X}}]\overline{T}, \\ & \qquad\quad \overline{b} \lessdot [\overline{b}/\overline{\texttt{Y}}][\overline{a}/\overline{\texttt{X}}]\overline{\texttt{P}}, \overline{a} \lessdot [\overline{a}/\overline{\texttt{X}}]\overline{\texttt{N}} \mid \overline{a}, \overline{b} \text{ fresh}\} \\ & \qquad \mid (\texttt{C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>.m} : \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\ \overline{T} \to T) \in \Pi\} \\ \quad\textbf{in} & (a, (C_R \cup \bigcup_i C_i \cup \{c\}))\end{array}$$

The `new`-expression is comparatively simple. Starting from a generic instance of the class type, we calculate the types $\overline{T}$ of the fields, impose subtyping constraints on the constructor argument $\overline{R}$, and check the subtyping constraints of the class.

$$\begin{array}{ll}\textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{new C}(\overline{\texttt{e}})) = & \\ \quad\textbf{let} & \forall \texttt{e}_i \in \overline{\texttt{e}} : (R_i, C_i) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}_i) \\ & \overline{a} \text{ fresh} \\ & \mathit{fields}(\texttt{C<}\overline{a}\texttt{>}) = \overline{T}\ \overline{\texttt{f}} \\ & C = \{\overline{R} \lessdot \overline{T}\} \cup \{\overline{a} \lessdot [\overline{a}/\overline{\texttt{X}}]\overline{\texttt{N}}\} \qquad \text{where } \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>}\{\ldots\} \\ \quad\textbf{in} & (\texttt{C<}\overline{a}\texttt{>}, C \cup \bigcup_i C_i)\end{array}$$

For cast expressions, we ignore the return type and pass on the constraints for the subexpression. We return the target type of the cast.

$$\begin{aligned}&\textbf{TYPEExpr}((\Pi;\overline{\eta}), (\texttt{N})\texttt{e}) = \\ &\qquad \textbf{let} \quad (R, C) = \textbf{TYPEExpr}((\Pi;\overline{\eta}), \texttt{e}) \\ &\qquad \textbf{in} \quad (\texttt{N}, C)\end{aligned}$$

▶ **Example 6.** To illustrate the constraint generation step we will apply it to the program depicted in figure 1b. First the **FJType** function assigns the fresh type variable f to the parameter `fst`. Afterwards the **TYPEExpr** function is called on the return expression of the `setfst` method. The local variable `fst` does not emit any constraints. For the `this.snd` part of the expression the **TYPEExpr** function returns an or-constraint:

$$\begin{aligned}\overline{c}_1 &= \textbf{TYPEExpr}((\Pi;\overline{\eta}), \texttt{this.snd}) \\ &= (b, oc(\{(\texttt{Pair<X, Y>} \lessdot \texttt{Pair<}w, y\texttt{>}), (b \doteq y), (w \lessdot \texttt{Object}), (y \lessdot \texttt{Object})\}))\end{aligned}$$

This constraint is merged with the constraints generated by the `new Pair` constructor call:

$$\begin{aligned}&\textbf{TYPEExpr}((\Pi;\overline{\eta}), \texttt{new Pair(fst, this.snd)}) \\ &\quad = (\texttt{Pair<}d, e\texttt{>}, \{(f \lessdot d), (b \lessdot e), (d \lessdot \texttt{Object}), (e \lessdot \texttt{Object})\} \cup \overline{c}_1)\end{aligned}$$

5 Constraint Solving

This section describes the **Unify** algorithm which is used to find solutions for the constraints generated by **FJType**.

It first attempts to transform a constraint set into solved form and reads off a solution in the form of a substitution.

▶ **Definition 7** (Solved form). *A set C of constraints is in solved form if it only contains constraints of the following form:*

1. $a \lessdot b$
2. $a \doteq b$,
3. $a \lessdot \texttt{C<}\overline{T}\texttt{>}$,
4. $a \doteq \texttt{C<}\overline{T}\texttt{>}$, *with* $a \notin \overline{T}$.

In case 3 and 4 the type variable a does not appear on the left of another constraint of the form 3 or 4.

For brevity, we write $a_0 \lessdot^* a_n$ for a non-empty chain of subtyping constraints between type variables $a_0 \lessdot a_1, a_1 \lessdot a_2, \ldots, a_{n-1} \lessdot a_n$ where $n > 0$.

5.1 Algorithm Unify(C, Δ)

The input of the algorithm is a set of constraints C and a type environment Δ. The type environment binds the generic type variables $\overline{\texttt{X}}$ to their upper bounds. It is used in invocations of the subtyping judgment.

The treatment of the generic class variables $\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}$ deserves some explanation. The algorithm must not substitute for these variables. Instead it treats them like parameterless abstract classes `Xi<>` which are subtypes of their respective $\texttt{N}_i$ (where the variable name $\texttt{X}_i$ is now treated like a class name). Example 8 illustrates this approach.

The first step of the algorithm eliminates or-constraints from constraint set C. To do so, we consider all combinations of selecting simple constraints from or-constraints in C. In general, we have that $C = \{\overline{sc}, oc_1, \ldots, oc_n\}$ and we execute the remaining steps for all $C' = \{\overline{sc}\} \cup \{\overline{sc}_1\} \cup \cdots \cup \{\overline{sc}_n\}$ where $\overline{sc}_i \in oc_i$.

Step 1. We apply the rules in Figures 16 and 17 exhaustively to C'.

Step 2. At this point, all constraints $sc \in C'$ are either in solved form or one of the following cases applies:

1. $\{\texttt{C<}\overline{T}\texttt{>} \lessdot \texttt{D<}\overline{U}\texttt{>}\} \subseteq C'$ where $\forall \overline{\texttt{X}}, \overline{\texttt{N}} \colon \Delta \not\vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$ (roughly, `C` cannot be a subtype of `D`) – in this case C' has no solution;
2. $\{a \lessdot \texttt{C<}\overline{T}\texttt{>}, a \lessdot \texttt{D<}\overline{V}\texttt{>}\} \subseteq C'$ where $\forall \overline{\texttt{X}}, \overline{\texttt{N}} \colon \Delta \not\vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$ and $\forall \overline{\texttt{X}}, \overline{\texttt{N}} \colon \Delta \not\vdash \texttt{D<}\overline{\texttt{X}}\texttt{>} <: \texttt{C<}\overline{\texttt{N}}\texttt{>}$ (roughly, `C` and `D` are not subtype-related) – in this case C' has no solution; or
3. $\{\texttt{C<}\overline{T}\texttt{>} \lessdot b\} \subseteq C'$.

The last case is a lower bound constraint which is embraced by Scala, but which is not legal in FGJ (nor in Java). As we insist on inferring a type, we have to find a concrete instance for $\texttt{C<}\overline{T}\texttt{>}$. To do so, we generate an or-constraint from each lower bound constraint and its corresponding upper bound constraint (using upper bound `Object` if no such constraint exists) as follows:

$$
\begin{aligned}
\mathit{expandLB}(\texttt{C<}\overline{T}\texttt{>} \lessdot b, b \lessdot \texttt{D<}\overline{U}\texttt{>}) = \{\{b \doteq [\overline{T}/\overline{\texttt{X}}]\texttt{N}\} \mid \Delta \vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{N}, \Delta \vdash \texttt{N} <: \texttt{D<}\overline{\texttt{P}}\texttt{>}\} \\
\text{where } \overline{\texttt{P}} \text{ is determined by } \Delta \vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{P}}\texttt{>} \text{ and } [\overline{T}/\overline{\texttt{X}}]\overline{\texttt{P}} = \overline{U}
\end{aligned}
$$

This constraint replaces the lower and upper bound constraint from which it was generated.

A lower bound may also be implied by a constraint set with constraints of the form $C_{ab} = a \lessdot \texttt{C<}\overline{T}\texttt{>}, a \lessdot^* b$. In this case $\texttt{C<}\overline{T}\texttt{>}$ must either be a upper or lower bound for b. We implement it by *expandLB*, which adds a lower bound constraint for b and also adding a upper bound to b. While C_{ab} remains in the constraint set: $\mathit{expandLB}(\texttt{C<}\overline{T}\texttt{>} \lessdot b, b \lessdot \texttt{D<}\overline{U}\texttt{>}) \cup \{b \lessdot \texttt{C<}\overline{T}\texttt{>}\}$

Now we are in a similar situation as before. Our current constraint set C' is a mix of simple constraints and or-constraints and, again, we consider all (simple) constraint sets C'' that arise as combinations of selecting simple constraints from C'.

Step 3. We apply the rule (subst) exhaustively to C'':

$$
\text{(subst)} \quad \frac{C \cup \{a \doteq T\}}{[T/a]C \cup \{a \doteq T\}} \quad a \text{ occurs in } C \text{ but not in } T
$$

We fail if we find any $a \doteq T$ such that a occurs in T.

Step 4. If C'' has changed from applying (subst), we continue with C'' from step 1.

Step 5. Otherwise, C'' is in solved form and it remains to eliminate subtyping constraints between variables by exhaustive application of rule (sub-elim) and (erase) (see Figure 17). Applying this rule does not affect the solve form property.

$$
\text{(sub-elim)} \quad \frac{C \cup \{a \lessdot b\}}{[a/b]C \cup \{b \doteq a\}}
$$

Step 6. We finish by generating a solving substitution from the remaining $\doteq$-constraints and generic variable declarations from the remaining $\lessdot$-constraints. Let $C'' = C_{\doteq} \cup C_{\lessdot}$ such that $C_{\doteq}$ contains only $\doteq$-constraints and $C_{\lessdot}$ contains only $\lessdot$-constraints. Now $C_{\lessdot} = \{\overline{a} \lessdot \overline{N}\}$ and choose some fresh generic variables $\overline{\texttt{Y}}$ of the same length as $\overline{a}$. We can read off the substitution σ from $C_{\doteq}$ where we need to substitute the generic variables for the type variables. We obtain the generic variable declarations directly from $C_{\lessdot}$ using the same generic variable substitution. We need not apply σ here because we applied (subst) exhaustively in Step 3.

$$
\begin{aligned}
\sigma &= \{b \mapsto [\overline{\texttt{Y}}/\overline{a}]T \mid (b \doteq T) \in C_{\doteq}\} \cup \{\overline{a} \mapsto \overline{\texttt{Y}}\} \cup \{b \mapsto \texttt{X} \mid (b \lessdot \texttt{X}) \in C_{\lessdot}\}, \\
\gamma &= \{\texttt{Y} \triangleleft [\overline{\texttt{Y}}/\overline{a}]N \mid (a \lessdot N) \in C_{\lessdot}\}
\end{aligned}
$$

We return the pair (σ, γ).

$$\text{(match)}\quad \frac{C \cup \{a \lessdot \texttt{C<}\overline{T}\texttt{>},\, a \lessdot \texttt{D<}\overline{V}\texttt{>}\}}{C \cup \{a \lessdot \texttt{C<}\overline{T}\texttt{>},\, \texttt{C<}\overline{T}\texttt{>} \lessdot \texttt{D<}\overline{V}\texttt{>}\}} \quad \Delta \vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$$

$$\text{(adopt)}\quad \frac{C \cup \{a \lessdot \texttt{C<}\overline{T}\texttt{>},\, b \lessdot^* a,\, b \lessdot \texttt{D<}\overline{U}\texttt{>}\}}{C \cup \{a \lessdot \texttt{C<}\overline{T}\texttt{>},\, b \lessdot^* a,\, b \lessdot \texttt{D<}\overline{U}\texttt{>},\, b \lessdot \texttt{C<}\overline{T}\texttt{>}\}}$$

$$\text{(adapt)}\quad \frac{C \cup \{\texttt{C<}\overline{T}\texttt{>} \lessdot \texttt{D<}\overline{U}\texttt{>}\}}{C \cup \{\texttt{D<}[\overline{T}/\overline{\texttt{X}}]\overline{\texttt{N}}\texttt{>} \doteq \texttt{D<}\overline{U}\texttt{>}\}} \quad \Delta \vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$$

$$\text{(reduce)}\quad \frac{C \cup \{\texttt{D<}\overline{T}\texttt{>} \doteq \texttt{D<}\overline{U}\texttt{>}\}}{C \cup \{\overline{T} \doteq \overline{U}\}}$$

$$\text{(equals)}\quad \frac{C \cup \{a_1 \lessdot a_2, a_2 \lessdot a_3, \ldots, a_n \lessdot a_1\}}{C \cup \{a_1 \doteq a_2, a_2 \doteq a_3, \ldots\}} \quad n > 0$$

Figure 16 Reduce and adapt rules.

$$\text{(erase)}\quad \frac{C \cup \{a \doteq a\}}{C}$$

$$\text{(swap)}\quad \frac{C \cup \{N \doteq a\}}{C \cup \{a \doteq N\}}$$

Figure 17 Erase and swap rules.

► **Example 8.** To illustrate our treatment of generic variables, we consider a typical case involving the (adapt) rule from Figure 16.

Consider $C = \{\texttt{X} \lessdot \texttt{D<}\overline{U}\texttt{>}\}$ and let $\texttt{X<: C<}\overline{\texttt{T}}\texttt{>} \in \Delta$ be the bound for `X`.

The side condition of the rule (adapt) asks for some $\overline{\texttt{N}}$ such that $\Delta \vdash \texttt{X<: D<}\overline{\texttt{N}}\texttt{>}$, i.e., "is there a way that `X` can be a subtype of `D`?"

By inversion of subtyping and transitivity, this judgment holds if $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{><: D<}\overline{\texttt{N}}\texttt{>}$ holds.

Hence, applying (adapt) to C yields $\{\texttt{D<}\overline{\texttt{N}}\texttt{>} \doteq \texttt{D<}\overline{U}\texttt{>}\}$. The substitution in the rule is empty because `X` is considered a parameterless type.

The remaining rules work similarly. In particular, different variables $\texttt{X} \neq \texttt{Y}$ give rise to different (abstract) classes. For example, the (reduce) rule removes the constraint $\texttt{X} \doteq \texttt{X}$, but it does *not* apply to $\texttt{X} \doteq \texttt{Y}$. Rather, an equation like this renders the constraint set unsolvable.

► **Example 9.** To see that the algorithm is able to infer bounded generic types, we consider the example from the introduction (Figure 1a). The `setfst` method obtains a new generic type variable `Z`. We clarify this via a counterexample. Let's assume in the example in Figure 1b that there is an additional method call `setfst(new Integer())` inside the class `Pair`. This call would cause the method `setfst` to obtain type `Pair<Integer, Y> setfst(Integer fst)`, which corresponds to the typing rules (c.f. `GT-CLASS` in Figure 9): Inside the same class methods cannot be used in a polymorphic way. We have to make this restriction to avoid polymorphic recursion.

6 Properties of Unify

▶ **Theorem 10** (Soundness). *If* $\textbf{\textit{Unify}}(C, \Delta) = (\sigma, \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}})$*, then* σ *is unifier of* $(C, \Delta \cup \{\overline{\mathtt{Y}} <: \overline{\mathtt{P}}\})$.

▶ **Theorem 11** (Completeness). $\textbf{\textit{Unify}}(C, \Delta)$ *calculates the set of general unifiers for* (C, Δ).

A set of general unifiers can provide any unifier as a substitution instance of one of its members.

▶ **Definition 12** (Set of general unifiers). *Let* C *be a set of constraints and* Δ *a type environment.*

A set of unifiers M *for* (C, Δ) *is called* set of general unifiers *if for any unifier* ω *for* (C, Δ) *there is some unifier* $\sigma \in M$ *and a substitution* λ *such that* $\omega = \lambda \circ \sigma$.

7 Soundness, completeness and complexity of type inference

After showing that type unification is sound and complete, we can now show that type inference **FJTypeInference** also is sound and complete.

▶ **Theorem 13** (Soundness). *For all* Π, $\mathtt{L}$, Π', $\textbf{\textit{FJTypeInference}}(\Pi, L) = \Pi'$ *implies* $\Pi \vdash L : \Pi'$.

▶ **Theorem 14** (Completeness). *For all* Π, $\mathtt{L}$, Π', $\Pi \vdash L : \Pi'$ *implies there is a* Π'' *with* $\textbf{\textit{FJTypeInference}}(\Pi, L) = \Pi''$, $\Pi \vdash L : \Pi''$*, and the types of* Π' *are instances of* Π''.

▶ **Theorem 15** (NP-Hardness). *The type inference algorithm for typeless Featherweight Java is NP-hard.*

▶ **Theorem 16** (NP-Completeness). *The type inference algorithm for typeless Featherweight Java is NP-Complete.*

8 Related Work

8.1 Formal models for Java

There is a range of formal models for Java. Flatt et al [7] define an elaborate model with interfaces and classes and prove a type soundness result. They do not address generics. Igarashi et al [11] define Featherweight Java and its generic sibling, Featherweight Generic Java. Their language is a functional calculus reduced to the bare essentials, they develop the full metatheory, they support generics, and study the type erasing transformation used by the Java compiler. MJ [4] is a core calculus that embraces imperative programming as it is targeted towards reasoning about effects. It does not consider generics. Welterweight Java [17] and OOlong [5] are different sketches for a core language that includes concurrency, which none of the other core languages considers.

We chose to base our development on FGJ because it embraces a relevant subset of Java without including too much complexity (e.g., no imperative features, no interfaces, no concurrency). It seems that results for FGJ are easily scalable to full Java. We leave the addition of these feature to future work, as we see our results on FGJ as a first step towards a formalized basis for global type inference for Java.

8.2 Type inference

Some object-oriented languages like Scala, C#, and Java perform *local* type inference [16,18]. Local type inference means that missing type annotations are recovered using only information from adjacent nodes in the syntax tree without long distance constraints. For instance, the type of a variable initialized with a non-functional expression or the return type of a method can be inferred. However, method argument types, in particular for recursive methods, cannot be inferred by local type inference.

Milner's algorithm $\mathcal{W}$ [15] is the gold standard for global type inference for languages with parametric polymorphism, which is used by ML-style languages. The fundamental idea of the algorithm is to enforce type equality by many-sorted type unification [14,25]. This approach is effective and results in so-called principal types because many-sorted unification is unitary, which means that there is at most one most general result.

Plümicke [20] presents a first attempt to adopt Milner's approach to Java. However, the presence of subtyping means that type unification is no longer unitary, but still finitary. Thus, there is no longer a single most general type, but any type is an instance of a finite set of maximal types (for more details see Section 8.3). Further work by the same author [22,24], refines this approach by moving to a constraint-based algorithm and by considering lambda expressions and Scale-like function types. In Plümicke's work there is no formal definition of the type system as a basis of the type inference algorithm. One contribution of this paper is a formal definition of the underlying type system.

We rule out polymorphic recursion because its presence makes type inference (but not type checking: see FGJ) undecidable. Henglein [9] as well as Kfoury et al [12] investigate type inference in the presence of polymorphic recursion. They show that type inference is reducible to semi-unification, which is undecidable [13]. However, the undecidability of this problem apparently does not matter much in practice [6].

Ancona, Damiani, Drossopoulou, and Zucca [1] consider polymorphic byte code. Their approach is modular in the sense that it infers polymorphic structural types. As Java does not support structural types, their approach would have to be simulated with generated interfaces. Plümicke [23] follows this approach. Furthermore Ancona and coworkers do not consider generic classes.

8.3 Unification

We reduce the type inference problem to constraint solving with equality and subtype constraints. The procedure presented in Section 5 is inspired by polymorphic order-sorted unification which is used in logic programming languages with polymorphic order-sorted types [2,8,10,27].

Smolka's thesis [27] mentions type unification as an open problem. He gives an incomplete type inference algorithm for the logical language TEL. The reason for incompleteness is the admission of subtype relationships between polymorphic types of different arities as in `List(a)`$<:$ `myLi(a,b)`. In consequence, the subtyping relation does not fulfill the ascending chain condition. For example, given `List(a)`$<:$ `myLi(a,b)`, we obtain:

`List(a)`$<:$ `myLi(a,List(a))`$<:$ `myLi(a,myLi(a,List(a)))`$<:$ $\ldots$

However, this subtyping chain exploits covariant subtyping, which does not apply to FGJ.

Smolka's algorithm also fails sometimes in the absence of infinite chains, although there is a unifier. For example, given `nat`$<:$ `int` and the set of subtyping constraints $\{\texttt{nat} \lessdot \texttt{a}, \texttt{int} \lessdot \texttt{a}\}$, it returns the substitution $\{\texttt{a} \mapsto \texttt{nat}\}$ generated from the first constraint encountered. This

substitution is not a solution because $\{\texttt{int} \lessdot \texttt{nat}\}$ fails. However, $\{\texttt{a} \mapsto \texttt{int}\}$ is a unifier, which can be obtained by processing the constraints in a different order: from $\{\texttt{int} \lessdot \texttt{a}, \texttt{nat} \lessdot \texttt{a}\}$ the algorithm calculates the unifier $\{\texttt{a} \mapsto \texttt{int}\}$.

Hill and Topor [10] propose a polymorphically typed logic programming language with subtyping. They restrict subtyping to type constructors of the same arity, which guarantees that all subtyping chains are finite. In this approach a *most general type unifier (mgtu)* is defined as an upper bound of different principal type unifiers. In general, two type terms need not have an upper bound in the subtype ordering, which means that there is no mgtu in the sense of Hill and Topor. For example, given $\texttt{nat} <: \texttt{int}$, $\texttt{neg} <: \texttt{int}$, and the set of inequations $\{\texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a}\}$, the mgtu $\{\texttt{a} \mapsto \texttt{int}\}$ is determined. If the subtype ordering is extended by $\texttt{int} <: \texttt{index}$ and $\texttt{int} <: \texttt{expr}$, then there are three unifiers $\{\texttt{a} \mapsto \texttt{int}\}$, $\{\texttt{a} \mapsto \texttt{index}\}$, and $\{\texttt{a} \mapsto \texttt{expr}\}$, but none of them is an mgtu [10].

The type system of PROTOS-L [2] was derived from TEL by disallowing any explicit subtype relationships between polymorphic type constructors. Beierle [2] gives a complete type unification algorithm, which can be extended to the type system of Hill and Topor. They also prove that the type unification problem is finitary.

Given the declarations $\texttt{nat} <: \texttt{int}$, $\texttt{neg} <: \texttt{int}$, $\texttt{int} <: \texttt{index}$, and $\texttt{int} <: \texttt{expr}$, applying the type unification algorithm of PROTOS-L to the set of inequations $\{\texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a}\}$ yield three general unifiers $\{\texttt{a} \mapsto \texttt{int}\}$, $\{\texttt{a} \mapsto \texttt{index}\}$, and $\{\texttt{a} \mapsto \texttt{expr}\}$.

Plümicke [21] realized that the type system of TEL is related to subtyping in Java. In contrast to TEL, where the ascending chain condition does not hold, Java with wildcards violates the descending chain condition. For example, given $\texttt{myLi<b,a>} <: \texttt{List<a>}$ we find:

$$\ldots <: \texttt{myLi<?}\triangleleft\texttt{myLi<?}\triangleleft\texttt{List<a>,a>,a>} <: \texttt{myLi<?}\triangleleft\texttt{List<a>,a>} <: \texttt{List<a>}$$

Plümicke [21] solved the open problem of infinite chains posed by Smolka [27]. He showed that in any infinite chain there is a finite number of elements such that all other elements of the chain are instances of them. The resulting type unification algorithm can be used for type inference of Java 5 with wildcards [20]. As FGJ has no wildcards, we based our algorithm on an earlier work [19]. In contrast to that work, which only infers generic methods with unbounded types, our algorithm infers bounded generics. To this end, we do not expand constraints of the form $a \lessdot N$, where a is type variable and N is is a non-variable type, but convert them to bounded type parameters of the form `X extends N`. This change results in a significant reduction of the number of solutions of the type unification algorithm without restricting the generality of typings of FGJ-programs. Unfortunately, constraints of the form $N \lessdot a$ have to be expanded as FGJ (like Java) does not permit lower bounds for generic parameters. If lower bounds were permitted (as in Scala), the number of solutions could be reduced even further.

9 Conclusions

This paper presents a global type inference algorithm applicable to Featherweight Generic Java (FGJ). To this end, we define a language FGJ-GT that characterizes FGJ programs amenable to type inference: its methods carry no type annotations and it does not permit polymorphic recursion. This language corresponds to a strict subset of FGJ.

The inference algorithm is constraint based and is able to infer generalized method types with bounded generic types, as demonstrated with the example in Figure 1a.

In future work, we plan to extend FGJ-GT to a calculus with wildcards inspired by Wild FJ [29]. We also plan to extend the formal calculus with lambda expressions (cf. [3]), but using true function types in place of interface types.

References

1 Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: compositional compilation for java-like languages. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 26–37. ACM, 2005. doi:10.1145/1040305.1040308.

2 Christoph Beierle. Type inferencing for polymorphic order-sorted logic programs. In *International Conference on Logic Programming*, pages 765–779, 1995.

3 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Venneri Betti. Java & lambda: A featherweight story. *Logical Methods in Computer Science*, 14(3:17):1–24, 2018.

4 G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, April 2003. doi:10.48456/tr-563.

5 Elias Castegren and Tobias Wrigstad. OOlong: an extensible concurrent object calculus. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1022–1029. ACM, 2018. doi:10.1145/3167132.3167243.

6 Martin Emms and Hans Leiß. Extending the type checker of standard ML by polymorphic recursion. *Theor. Comput. Sci.*, 212(1-2):157–181, 1999. doi:10.1016/S0304-3975(98)00139-X.

7 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269. Springer, 1999. doi:10.1007/3-540-48737-9_7.

8 Michael Hanus. Parametric order-sorted types in logic programming. *Proc. TAPSOFT 1991*, LNCS(394):181–200, 1991.

9 Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993. doi:10.1145/169701.169692.

10 Patricia M. Hill and Rodney W. Topor. A Semantics for Typed Logic Programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

11 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

12 A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993. doi:10.1145/169701.169687.

13 A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Inf. Comput.*, 102(1):83–101, 1993. doi:10.1006/inco.1993.1003.

14 A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

15 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

16 Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. *Proc. 28th ACM Symposium on Principles of Programming Languages*, 36(3):41–53, 2001.

17 Johan Östlund and Tobias Wrigstad. Welterweight java. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 97–116. Springer, 2010. doi:10.1007/978-3-642-13953-6_6.

18 Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 252–265, 1998.

19 Martin Plümicke. Type Unification in Generic–Java. In Michael Kohlhase, editor, *Proceedings of 18th International Workshop on Unification (UNIF'04)*, Cork, July 2004.

20 Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

21 Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

22 Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

23 Martin Plümicke. Structural type inference in java-like languages. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016.*, pages 109–113, 2016. URL: http://ceur-ws.org/Vol-1559/paper09.pdf.

24 Martin Plümicke and Andreas Stadelmeier. Introducing Scala-like function types into Java-TX. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 23–34, New York, NY, USA, 2017. ACM. doi:10.1145/3132190.3132203.

25 J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, January 1965.

26 Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In Atsushi Ohori, editor, *Programming Languages and Systems, First Asian Symposium, APLAS 2003*, volume 2895 of *Lecture Notes in Computer Science*, pages 283–302, Beijing, China, November 2003. Springer. doi:10.1007/978-3-540-40018-9_19.

27 Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany, May 1989.

28 Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. Global type inference for Featherweight Generic Java, 2022. URL: https://arxiv.org/abs/2205.08768.

29 Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In Philip Wadler, editor, *Proceedings of FOOL 12*, Long Beach, California, USA, January 2005. ACM, School of Informatics, University of Edinburgh. URL: http://homepages.inf.ed.ac.uk/wadler/fool/.

Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BackREST

François Gauthier ✉
Oracle Labs, Brisbane, Australia

Behnaz Hassanshahi ✉
Oracle Labs, Brisbane, Australia

Benjamin Selwyn-Smith ✉
Oracle Labs, Brisbane, Australia

Trong Nhan Mai ✉
Oracle Labs, Brisbane, Australia

Max Schlüter ✉
Oracle Labs, Brisbane, Australia

Micah Williams ✉
Oracle, Durham, NC, USA

Abstract

Following the advent of the American Fuzzy Lop (AFL), fuzzing had a surge in popularity, and modern day fuzzers range from simple blackbox random input generators to complex whitebox concolic frameworks that are capable of deep program introspection. Web application fuzzers, however, did not benefit from the tremendous advancements in fuzzing for binary programs and remain largely blackbox in nature. In this experience paper, we show how techniques like state-aware crawling, type inference, coverage and taint analysis can be integrated with a black-box fuzzer to find *more* critical vulnerabilities, *faster* (speedups between 7.4× and 25.9×). Comparing BackREST against three other web fuzzers on five large (>500 KLOC) Node.js applications shows how it consistently achieves comparable coverage while reporting more vulnerabilities than state-of-the-art. Finally, using BackREST, we uncovered eight 0-days, out of which six were not reported by any other fuzzer. All the 0-days have been disclosed and most are now public, including two in the highly popular Sequelize and Mongodb libraries.

2012 ACM Subject Classification Security and privacy → Web application security

Keywords and phrases Taint analysis, fuzzing, crawler, Node.js

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.29

Related Version *Previous Version*: `https://arxiv.org/abs/2108.08455`

Supplementary Material *Software (Source Code)*: `https://github.com/uqcyber/NodeJSFuzzing`
archived at `swh:1:dir:7282fcd2dbf1052a792609716114557308 5c4487`

1 Introduction

Fuzzing encompasses techniques and tools to automatically identify vulnerabilities in programs by sending malformed or malicious inputs and monitoring abnormal behaviours. Nowadays, fuzzers come in three major shades: blackbox, greybox, and whitebox, according to how much visibility they have into the program internals [42]. Greybox fuzzing, which was made popular by the AFL fuzzer, combines blackbox with lightweight whitebox techniques, and have proven to be very effective at fuzzing programs that operate on binary input [107, 26, 25].

Most web application fuzzers that are used in practice are still blackbox [4, 7, 2, 79], and, despite decades of development, still struggle to automatically detect well studied vulnerabilities such as SQLi, and XSS [84]. As a result, security testing teams have to invest significant manual efforts into building models of the application and driving the fuzzer to

© François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams;
licensed under Creative Commons License CC-BY 4.0

trigger vulnerabilities. To overcome the limitations of current blackbox web application fuzzers, and find more vulnerabilities automatically, new strategies must be investigated. To pave the way for the next generation of practical web application fuzzers, this paper first shows how REST-like API models are a suitable abstraction for web applications and how adding lightweight coverage and taint feedback loops to a blackbox fuzzer can significantly improve performance and detection capabilities. Then, it highlights how the resulting BackREST greybox fuzzer consistently detects more vulnerabilities than state-of-the-art. Finally, our evaluation reveals how BackREST found eight 0-days, out of which six were *missed* by all the web application fuzzers we compared it against.

API model inference. Model-based fuzzers, which use a model to impose *constraints* on input, dominate the web application fuzzing scene. Existing model-based web application fuzzers typically use dynamically captured traffic to derive a base model, which can be further enhanced manually [5, 2, 4]. As with any dynamic analysis, relying on captured traffic makes the quality of the model dependent on the quality of the traffic generator, be it a human being, a test suite, or a crawler. To navigate JavaScript-heavy applications and trigger a maximum number of endpoints, BackREST directs a state-aware crawler towards JavaScript functions that trigger server-side interactions. For completeness, a static type inference analysis is then used to complement the dynamic model.

Feedback-driven. What makes greybox fuzzers so efficient is the feedback loop between the lightweight whitebox analysis components, and the blackbox input generator. BackREST is the first web application fuzzer that can focus the fuzzing session on those areas of the application that have not been exercised yet (i.e. by using coverage feedback), and that have a higher chance of containing security vulnerabilities (i.e. by using taint feedback). Taint feedback in BackREST further reports the *type* (e.g. SQLi, XSS, or command injection) of potential vulnerabilities, enabling BackREST to aggressively fuzz a given endpoint with vulnerability-specific payloads, yielding tremendous performance improvements in practice.

Validated in practice. The two main metrics that drive practical adoption of a fuzzer are the number of vulnerabilities it can detect and the time required to discover them. Our experiments show how BackREST uncovered eight 0-days in five large (> 500KLOC) Node.js applications, and how adding lightweight whitebox analyses significantly speeds up (7.4-25.9× faster) the fuzzing session. All 0-days have been disclosed, and most have been announced as NPM advisories, meaning that developers will be alerted about them when they update their systems. Four of them have been tagged with high or critical severity by independent third-parties, and two have been reported against the highly popular `sequelize` (648 745 weekly downloads) and `mongodb` (1 671 653 weekly downloads) libraries.

Contributions. This paper makes the following contributions:

- We show how REST-like APIs can effectively model the server-side of modern web applications, and quantify how coverage and taint feedback enhance coverage, performance, and vulnerability detection.
- We empirically evaluate BackREST on five large (>500 KLOC) Node.js (JavaScript) web applications; a platform and language that are notoriously difficult to analyse, and under-represented in current security literature.

- We compare BackREST against three state-of-the-art fuzzers (Arachni, Zap, and w3af) and open-source our test harness [1].
- We show how greybox fuzzing for web applications allows to detect *severe* 0-days that are missed by all the blackbox fuzzers we evaluated.

Takeaways. In our industrial setting, black-box web application fuzzing is used as a security testing tool, where the goal is to automatically detect a maximum number of security bugs and security regressions in a limited amount of time (e.g. a nightly test). With BackREST, we show that extending a black-box web application fuzzer with simple grey-box analyses that use coverage and taint feedback to *skip* and *select* rather than *derive* new inputs can reduce runtime while increasing the number of reported bugs. In our case, the investment in development time (i.e. to extend an existing black-box fuzzer) was quickly dwarfed by the time saved during each fuzzing campaign.

The rest of this paper is structured as follows. Section 2 presents our novel API model inference technique. Section 3 and Section 4 detail the BackREST feedback-driven fuzzing algorithm and implementation, respectively. Section 5 evaluates BackREST in terms of coverage, performance, and detected vulnerabilities and compares it against state-of-the-art web application fuzzers. Section 6 presents and explains reported 0-days. Sections 7 and 8 present related work and conclude the paper.

2 API Model Inference

Web applications expose entry points in the form of URLs that clients can interact with via the HTTP protocol. However, the HTTP protocol specifies only how a client and a server can send and receive information over a network connection, not how to structure the interactions. Nowadays, representational state transfer (REST) is the *de facto* protocol that most modern client-side applications use to communicate with their backend server. While the REST protocol was primarily aimed at governing interactions with web services, we make the fundamental observation that client-server interactions in modern web applications can also be modelled as REST-like APIs. Indeed, at its core, REST uses standard HTTP verbs, URLs, and request parameters to define and encapsulate client-server interactions, which, from our experience, is also what many modern web application frameworks do (e.g. Spring (Java), Ruby on Rails (Ruby), Django (Python), and Express.js (JavaScript)).

Despite the plethora of REST-related tools available, the task of creating an initial REST specification remains, however, largely manual. While tools exist to convert captured traffic into a REST specification, the burden of thoroughly exercising the application or augmenting the specification with missing information is still borne by developers. BackREST alleviates this manual effort by extending a state-aware crawler designed for rich client-side JavaScript applications [50] to dynamically infer REST APIs through crawling. Modern web applications, and single-page ones in particular, implement complex and highly interactive functionalities on the client side. A recent Stack Overflow developer survey [10] shows that web applications are increasingly built using complex client-side frameworks, such as AngularJS [8] and React [9]. In fact, three out of five applications we evaluate in Section 5 heavily use such frameworks. Using a state-aware crawler that can automatically navigate complex client-side frameworks allows BackREST to discover server-side endpoints that can only be triggered through complex JavaScript interactions.

[1] `https://github.com/uqcyber/NodeJSFuzzing`

2.1 Motivating Example

Listing 1 shows an endpoint definition from a Node.js Express application. At line 1, `app` refers to the programmatic web application object, and the `delete` method is used to define an HTTP `DELETE` entry point. The arguments to `delete` include the URL (i.e. `"/users/"`) and path parameter (i.e. `":userId"`) of the entry point, and the callback function that will be executed on incoming requests. The callback function receives request (i.e. `req`) and response (i.e. `res`) objects as arguments, reads the `userId` path parameter at line 2, and removes the corresponding document from a collection in the database at line 3, and leaves the response untouched. The API specification, in OpenAPI format [93], corresponding to the example entry point of Listing 1 is shown in Listing 2. Lines 2-4 define the `"paths"` entry that lists valid URL paths, where each path contains one entry per valid HTTP method. Lines 5-13 define the `"parameters"` object that lists the valid parameters for a given path and HTTP method. Specifically, line 7 defines the name of the parameter, line 8 specifies that it is a path parameter, line 9 specifies that the parameter is required, line 10 specifies that the expected type of the `"userId"` parameter is `"string"` and line 11 captures a concrete example value that was observed while crawling. Directing the crawler to exercise client-side code that will trigger server-side endpoints is, however, non-trivial and covered in the next section.

Listing 1 Example endpoint and its callback in Express

```
1 app.delete("/users/:userId", (req, res) => {
2   const id = req.params.userId;
3   collection.remove({"id": id});
4 });
```

Listing 2 Automatically generated OpenAPI specification for the endpoint in Listing 1

```
1 {
2   "paths": {
3     "/users/{userId}": {
4       "delete": {
5         "parameters": [
6           {
7             "name": "userId",
8             "in": "path",
9             "required": true,
10            "type": "string",
11            "example": "abc123"
12          }
13        ]
14      }
15    }
16  }
17 }
```

2.2 Prioritised State-Aware Crawling for API Inference

To improve responsiveness, modern web applications often transfer a large portion of their logic, including data pre-processing and validation to the client side. To prevent errors, frameworks implement checks on the server-side that validate the structure and to some extent the content of incoming requests. Fuzzing modern web applications thus requires us to produce requests that get past those initial server-side checks. To this end, BackREST

infers APIs from the requests generated by a state-aware crawler. Our crawler extends the one presented in [50] to automatically discover endpoints, parameters and types, and to produce concrete examples, as shown in Listing 2.

Figure 1 shows the architecture of the crawler in [50]. It combines prioritized link extraction (or spidering) and state-aware crawling to support both multi-page and single-page applications. It performs static link extraction and dynamic state processing using tasks that are managed in parallel. Given the URL of a running instance, the crawler's task manager first creates a link extraction task for the top level URL, where static links are extracted from HTML elements. Next, new link extraction tasks are created recursively for newly discovered links. At the same time, it creates new states for each extracted link and adds them to a priority queue in a state crawl task. The state queue is prioritized either in a Depth First Search (DFS) or Breadth First Search (BFS) order, depending on the structure of the application. For instance, if an application is a traditional multi-page application and most of the endpoints would be triggered through static links, BFS can be more suitable, whereas a single-page application with workflows involving long sequences of client-side interactions, such as filling input elements and clicking buttons, would benefit more from a DFS traversal.

A state in the crawler includes the URL of the loaded page, its DOM tree, static links and valid events. To avoid revisiting same states, it compares the URLs and their DOM trees using a given distance threshold. As described in [50], it marks a state as previously visited if there exists a state in the cache that has the same URL and a DOM tree that is similar enough to the existing state. To compare the DOM trees, it parses them using the ElementTree XML [11] library and considers two trees to be in the same equivalence class if the number of different nodes does not exceed a threshold.

The crawler transitions between states by automatically triggering events that it extracts from HTML elements. It supports both statically and dynamically registered events, as well as customized event registration in frameworks. Determining the priority of events is one of the differentiating factors between state-aware crawlers. On the one hand, Crawljax [73], a well-known crawler for AJAX-driven applications randomly selects events from a state. On the other hand, Artemis [20], selects events that are more likely to increase code coverage. Feedex [74] instead prioritizes events that trigger user-specified workflows, such as adding an item. jÄk [79] uses dynamic analysis to create a navigation graph with dynamically generated URLs and traces that contain runtime information for events. Its crawler then navigates the graph in an attempt to maximise server-side coverage. Similar to jÄk, we also perform dynamic analysis to detect dynamically registered events that are difficult to detect statically and maximise coverage of server-side endpoints.

Contrary to jÄk, however, our crawler also uses the JavaScript call graph analysis by Feldthaus et al. [35] to compute a distance metric from event handler functions to target functions like `XMLHttpRequest.send()`, `HTMLFormElement.submit()`, or `fetch` and prioritize events that have smaller distances. While the call graph is not sound nor precise, it is being refined as the application is being crawled, allowing the crawler to reach increasingly deeply embedded target functions in the application code and the frameworks and libraries it uses. This prioritization strategy allows BackREST to maximize coverage of JavaScript functions that trigger server-side endpoints. For more details about the refined JavaScript call graph and the prioritization strategy in the crawler see [50]. Listing 3 shows a hande-made and very simplified code-snippet that uses AngularJS for event handling. While the first button, `B1` is a normal button and uses standard `click` event registration, the second button, `B2` uses a custom event registration from the AngularJS framework. Our crawler can successfully correlate the customized `click` event for button `B2` and prioritize `B2` over `B1` to call `fetch("users/abc123")` and trigger the server-side `users/abc123` endpoint.

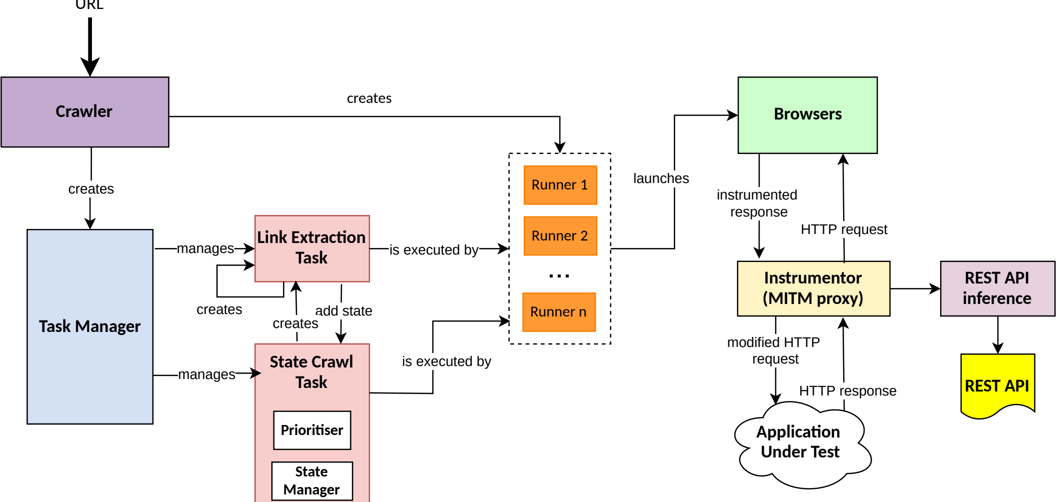


Figure 1 State-aware crawler architecture.

Listing 3 Client-side JavaScript code that uses a framework (Angular.js) with customized event handling and registration.

```
1  <html>
2    <script src='angular.js'></script>
3    <script>
4      function foo(){
5      }
6      function bar() {
7        fetch("users/abc123");
8      }
9    </script>
10   <body>
11     <button id='b1' click='foo()'>B1</button>
12     <button id='b2' data-ng-click='bar()'>B2</button>
13   </body>
14 </html>
15
```

One of the challenges in dynamic analysis of web applications is performing authentication and correctly maintaining the authenticated sessions. Our crawler provides support for a wide variety of authentication mechanisms, including Single Sign On, using a record-and-replay mechanism. We require the user to record the authentication once and use it to authenticate the application as many times as required. To verify whether a session is valid, we ask the user to provide an endpoint and pattern to look up in the response content once and before the crawling starts. For instance, for Juice-shop (see Section 5), we verify the session by sending a GET request to the `rest/user/whoami` endpoint and check if `"admin@juice-sh.op"` is present in its response content periodically to make sure it is logged in.

We intercept the requests triggered by our crawler using a Man-In-The-Middle (MITM) proxy. Next, we process the recorded HTTP requests to infer an API specification automatically. Because we use a client-side crawler to trigger the endpoints, the recorded traffic contains valid headers and parameter values that are persisted in the API and reused in the fuzzing phase. These seed values can often prove invaluable to get past server-side value checks. Our API inference also aggregates concrete values to infer their types. Going back to

the example in Listing 1, by automatically triggering `delete` requests to `/users/` endpoint with an actual `userId` string value (e.g. `abc123`), our API inference adds `userId` path parameter with `string` type to the specification based on the observed `userId` values.

2.2.1 Augmenting Crawled APIs with Static Type Inference

By definition, crawled APIs only capture those endpoints and parameters that were exercised dynamically, meaning that they typically under-approximate the real API of an application. We thus *optionally* complement crawled APIs with statically inferred endpoints and parameters. While static analysis can, in theory, over-approximate the real API of an application, precise static analysis of an entire web application stack is practically infeasible without the help of *stubs*, *mocks*, models, or over- and under-approximate clients [14, 61, 70, 101]. Our work is no exception, and we overcome the challenge of statically analysing `Node.js` web applications through the use of mock request and response objects, combined with an approximate use-based type inference analysis.

The inference analysis starts from application endpoints (see Listing 1), where it collects the declared endpoint URL and path parameters. It then initiates a use-based analysis that populates a mock request object with additional parameters that are *read* from the request, without being explicitly declared as path parameters. Once the mock request is populated with parameters, another analysis infers parameter types from their use.

We now illustrate the inner workings of the inference analysis by revisiting the example in Listing 1. The use-based analysis builds on top of a static type inference analysis [45] that approximates, in an unsound way, the runtime structural types of the elements in a program. In Figure 2, items on the left-hand side of the dashed line show the initial type propagation graph. Orange boxes represent types, grey circles represent abstract elements of the program, purple boxes represent function calls or field accesses, and black arrows capture the flow of types and data in the program. To help the reader map the elements of Figure 2 to the code of Listing 1, whenever possible, we annotated function types and field accesses with their code definition (e.g. `app.delete`, `req.params`). Starting from the top, the type propagation graph shows that `app.delete` is of type function and takes two parameters: a path of type string and a callback that is typed as a function that takes `req` and `res` objects as arguments. Then, `params` field of the `req` object is accessed, followed by the `userId` field that yields the `id` object. The rest of the graph is derived from the other statements in Listing 1. To collect request parameters and their types, our analysis propagates a mock request object through the type propagation graph at step 2., extends it with used parameters (i.e. parameters that are read from the request object) at step 3. and finally infer the types of used parameters at step 4. This final step uses and extends the type specifications of the `Tern.js` tool [51, 45]. The inferred parameters and types are then merged into crawled APIs.

3 Feedback-driven Fuzzing

BackREST builds on top of Sulley [16], a blackbox fuzzer with built-in networking support, and extends it with support for API parsing and fuzzing, as well as coverage and taint feedback. Figure 3 shows the high-level architecture of BackREST. First, the application under test is crawled, an API is derived (1), and augmented with statically inferred parameters (2). Then, BackREST is invoked on the generated API file (3). The API is then parsed (4) and broken down into low-level fuzzing code blocks (5). The type information included in the API file is used to select relevant mutation strategies (e.g. string, integer, JWT token, etc.).

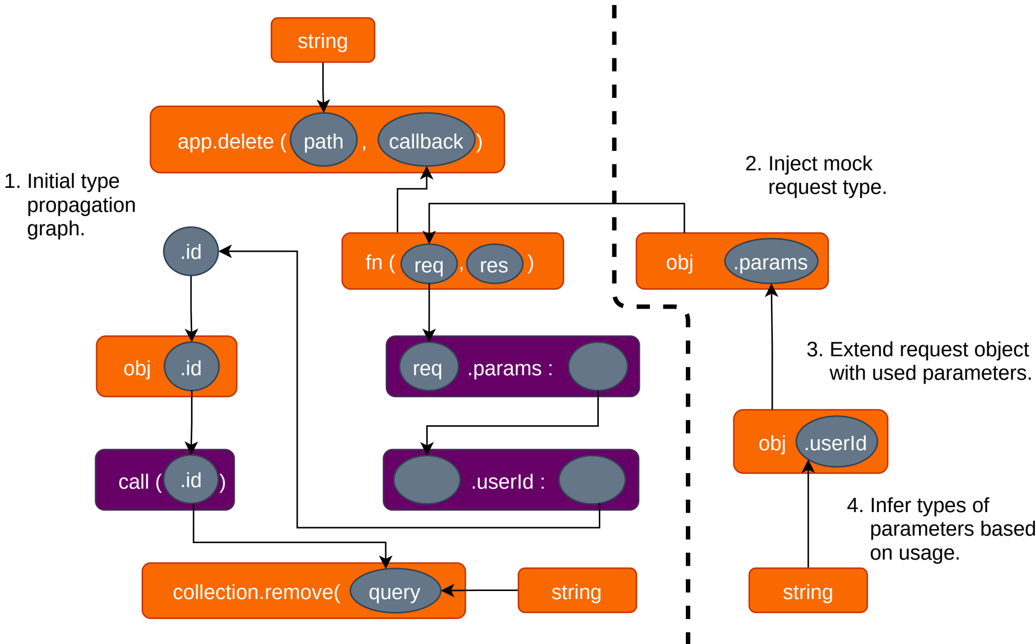


Figure 2 Use-based inference of request parameters and types.

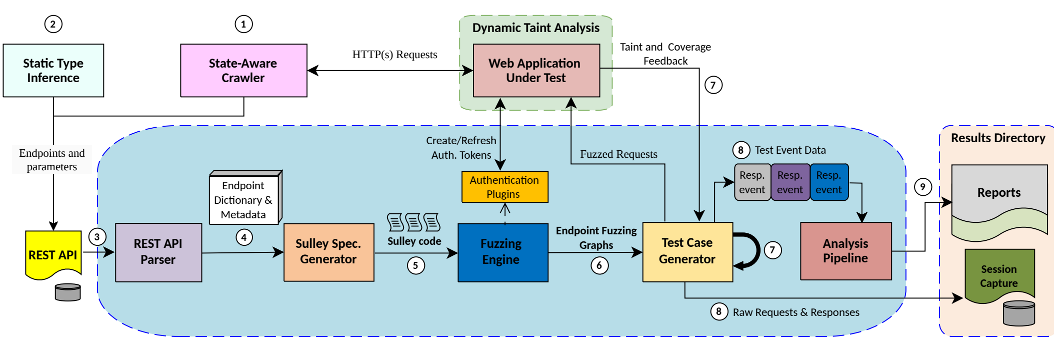


Figure 3 BackREST architecture.

The fuzzing engine is then responsible for evaluating the individual fuzzing code blocks and reassembling them into a graph that will yield well-formed HTTP requests (6). The test case generator repeatedly traverses the graph to generate concrete HTTP requests, sends them to the application under test, and monitors taint and coverage feedback (7). The HTTP responses are dispatched to the analysis pipeline, which runs in a separate thread, to detect exploits, and are also stored as-is for logging purposes (8). Indicators of exploitation include the response time, the error code and error messages, reflected payloads, and taint feedback. Finally, the analysis results are aggregated and reported (9). To simplify our evaluation setup (section 5), we run BackREST in *deterministic* mode, meaning that it always yields the same sequence of fuzzed HTTP requests for a given configuration.

3.1 Coverage Feedback

Coverage feedback, where the fuzzer uses online coverage information to guide the fuzzing session, was made popular by the AFL fuzzer [107]. Nowadays, most greybox fuzzers use coverage to guide their input generation engine towards producing input that will exercise newly covered code, or branches that will likely lead to new code [66, 65, 36, 80, 64, 26, 25]. Kelinci [57] ported AFL-style greybox fuzzing to Java programs by emulating AFL's coverage analysis and using the AFL fuzzing engine as-is. JQF [77] instead combines QuickCheck-style testing [27] and feedback-directed fuzzing to support inputs with arbitrary data structures. The underlying assumption in AFL and all its derivatives is that targeting code that has not been thoroughly exercised increases the likelihood of triggering bugs and vulnerabilities. Empirical evidence suggests that this assumption holds true for many codebases. Furthermore, the simplicity and widespread availability of coverage analysis makes it suitable for applications written in a wide range of languages.

Compared to mutation-based greybox fuzzers like AFL, BackREST uses coverage information differently. Where AFL-like fuzzers use coverage information to *derive* the next round of input, BackREST uses coverage information to *skip* inputs in the test plan that would likely exercise well-covered code. From that perspective, BackREST uses coverage information as a performance optimisation. Section 3.3 details how BackREST uses coverage information.

3.2 Taint Feedback

Taint-directed fuzzing, where the fuzzer uses taint tracking to locate sections of input that influence values at key program locations (e.g. buffer index, or magic byte checks), was pioneered by Ganesh et al. with the BuzzFuzz fuzzer [37]. In recent years, many more taint-directed greybox fuzzers that build on the ideas of BuzzFuzz have been developed [23, 67, 103, 104, 47, 86].

BackREST uses taint analysis in a different way. With the help of a lightweight dynamic taint inference analysis [41], it reports which input reaches security-sensitive program locations, and the type of vulnerability that could be triggered at each location. Armed with this information, BackREST can prioritise payloads that are more likely to trigger potential vulnerabilities. Taint feedback thus enables BackREST to *zoom in* payloads that are more likely to trigger vulnerabilities, which improves performance and detection capabilities. Taint feedback also improves detection capabilities in cases where exploitation cannot be easily detected in a blackbox manner. Finally, similar to coverage analysis, the relative simplicity of taint inference analysis makes it easy to port to a wide range of languages. The next section details how BackREST uses taint feedback during fuzzing.

3.3 BackREST Fuzzing Algorithm

Algorithm 1 shows the BackREST fuzzing algorithm. It first builds a test plan, based on the API model received as input (line 2). The test plan breaks the API model into a set of *endpoints*, lists "fuzzable" *locations* in each endpoint, and establishes a mutation schedule that specifies the values that are going to be injected at each location. Values are either cloned from the `example` fields, derived using mutations (omitted from Algorithm 1 for readability), or drawn from a pre-defined dictionary of payloads where vulnerability types map to a set of payloads. For example, the SQLi payload set contains strings like: `' OR '1'='1' -`, while the buffer overflow set contains very large strings and numbers.

Algorithm 1 API-based feedback-driven fuzzing.

```
   Input: web app. 𝒲, API model 𝒜, threshold 𝒯, payload dictionary 𝒟
   Output: vulnerability report 𝒱
 1 𝒫 ← BuildTestPlan(𝒜)
 2 𝒲′ ← coverageInstrument(𝒲)
 3 𝒲″ ← taintInstrument(𝒲′)
 4 totalCov ← 0
 5 foreach endpoint in 𝒫 do
 6   foreach location in 𝒫[endpoint] do
 7     types ← 𝒟.keys()
 8     taint:
 9     foreach type in types do
10       coverage:
11       count ← 0
12       foreach payload in 𝒟[type] do
13         (resp, currCov, taintCat) ← fuzz(endpoint, location, payload, 𝒲″)
14         count ← count + 1
15         if currCov > totalCov then
16           count ← 0
17         end
18         𝒱 ← 𝒱 ∪ DetectVulnerability(resp)
19         totalCov ← currCov
20         if taintCat ≠ ∅ then
21           types ← taintCat
22           if type ∉ types then
23             continue taint
24           end
25           count ← 0
26         end
27         if count > 𝒯 then
28           continue coverage
29         end
30       end
31     end
32   end
33 end
34 return 𝒱
```

Listing 4 Fuzzable locations for an example endpoint.

```
1 "/users/{userId}": {
2   "delete": {
3     "parameters": [
4       {
5         "name": "userId",
6         "in": "path",
7         "required": true,
8         "type": "string",
9         "example": "abc123"
10      }
11    ]
12  }
13 }
```

Listing 4 shows an example of an endpoint definition with fuzzable locations in bold. From top to bottom, fuzzable locations include: the value of the `userId` parameter, where the parameter will be injected (e.g. path or request body), whether the parameter is required or optional, and the type of the parameter. Other locations are left untouched to preserve the core structure of the model and increase the likelihood of the request getting past shallow syntactic and semantic checks.

Once the test plan is built, BackREST instruments the application for coverage and taint inference (lines 3-4 of Algorithm 1). Then, it starts iterating over the endpoints in the test plan (line 6). The fuzzer further sends a request for each `(endpoint, location, payload)` combination, collects the response, coverage and taint report, and increases its request counter by one (lines 13-15). If the request covers new code, the request counter is reset to zero, allowing the fuzzer to spend more time fuzzing that particular endpoint, location, and vulnerability type (lines 16-18). The vulnerability detector then inspects the response, searching for indicators of exploitation, and logs potential vulnerabilities (line 19).

Because blackbox vulnerability detectors inspect the response only, they might miss cases where an input reached a security-sensitive sink, without producing an observable side-effect. For example, a command injection vulnerability can be detected in a blackbox fashion only when an input triggers an observable side-effect, such as printing a fuzzer-controlled string, or making the application sleep for a certain amount of time. With taint feedback, however, the fuzzer is informed about: 1. whether parts of the input reached a sink, and 2. the vulnerability type associated with the sink. When the fuzzer is informed that an input reached a sink, it immediately *jumps* to the vulnerability type that matches that of the sink and starts sending payloads of that type only (lines 21-27). The idea behind this heuristic, which we validated on our benchmarks (see Subsection 5.3), is that payloads that match the sink type have a higher chance of triggering observable side-effects to help confirm a potential vulnerability. Targeting specific vulnerability types further minimises the number of inputs required to trigger a vulnerability. Finally, if a given endpoint and location pair have been fuzzed for more than $\mathcal{T}$ requests without increasing coverage, the fuzzer *jumps* to the next vulnerability type (lines 28-30). The idea behind this heuristic, which we also validated on our benchmarks, is that the likelihood of covering new code by fuzzing a given endpoint decreases with the number of requests, unless more complex techniques like symbolic execution are used. In our setup, we set $\mathcal{T}$ to 10 after trying out values in $\{0, 5, 10, 15, 20\}$ and keeping the minimal threshold that detected the maximum number of vulnerabilities. Promptly switching to payloads from a different vulnerability type reduces the total number of requests, thereby improving the overall performance.

Table 1 Benchmark applications.

Application	Description	Version	SLOC	Files
Nodegoat	Educational	1.3.0	970 450	12 180
Keystone	CMS	4.0.0	1 393 144	13 891
Apostrophe	CMS	2.0.0	774 203	5 701
Juice-shop	Educational	8.3.0	725 101	7 449
Mongo-express	DB manager	0.51.0	646 403	7 378

4 Implementation

BackREST brings together and builds on top of several existing components. The API inference component uses the state-aware crawler from [50] with an intercepting proxy [28] to generate and capture traffic dynamically. The static API inference uses `Tern.js` [51, 45] to perform type inference. The API parser is derived from PySwagger [6] and the fuzzing infrastructure extends Sulley [16], with API processing support. For coverage instrumentation, we use the Istanbul library [17] and have added a middleware in benchmark applications to read the coverage object after each request, and inject a custom header to communicate coverage results back to the fuzzer. For taint feedback, BackREST implements the Node.js taint analysis from [41] and extends our custom middleware to also communicate taint results back to the fuzzer. The taint analysis is itself built on top of the NodeProf.js instrumentation framework [97] that runs on the GraalVM[2] [3] [106] runtime.

5 Evaluation

In this section, we first review our experimental protocol. Next, we assess the contribution of static type inference to crawled APIs. then, we evaluate how coverage and taint feedback increase the coverage, performance, or number of vulnerabilities detected. The benchmark applications used for evaluation are listed in Table 1. All experiments were run on a machine with 8 Intel Xeon E5-2690 2.60GHz processors and 32GB memory. Then, we compare BackREST to three state-of-the-art web fuzzers. Finally, we present and explain the 0-days that BackREST detected.

5.1 Experimental Design

We took great care to design an empirical evaluation protocol that is fair and adequate. In this section, we review our protocol in the light of the SIGPLAN empirical evaluation guideline [3] and justify divergences from best practices. First, our benchmark applications are all implemented in Node.js and our results might not generalise to web applications implemented in other languages. As a rule of thumb, applications written in languages that have mature instrumentation frameworks will be more easily amenable to the kind of feedback-driven fuzzing implemented in BackREST. Next, whenever we compare BackREST either against itself or other fuzzers, unless otherwise stated (e.g. disabling of certain feedback loops), all runs of BackREST were parameterised in the exact same way. Benchmark-wise, the very

[2] `https://www.graalvm.org/`
[3] GraalVM is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Table 2 Total number of inferred endpoints and parameters. Number of statically inferred values in parenthesis.

Benchmark	# Entry points	# Request parameters
Nodegoat	19 (0)	28 (10)
Keystone	20 (0)	69 (46)
Apostrophe	184 (0)	633 (531)
Juice-shop	69 (0)	71 (64)
Mongo-express	29 (0)	96 (49)

Table 3 Impact of the coverage (C) and taint (T) feedback loops on runtime and total coverage, compared to baseline (B).

Benchmark	Coverage (%)			Time (hh:mm:ss)				
	B	C	CT	B	C		CT	
Nodegoat	80.31	78.54	75.59	0:42:39	0:06:07	(7.0×)	00:05:44	(7.4×)
Keystone	48.31	48.05	45.43	5:46:29	0:49:25	(7.0×)	0:13:23	(25.9×)
Apostrophe	—	48.40	45.52	—	11:11:42	–	6:17:34	–
Juice-Shop	74.73	76.34	75.85	12:48:15	1:10:31	(10.9×)	1:08:26	(11.2×)
Mongo-express	69.62	69.57	66.59	2:21:49	0:16:07	(8.8×)	0:11:07	(12.8×)

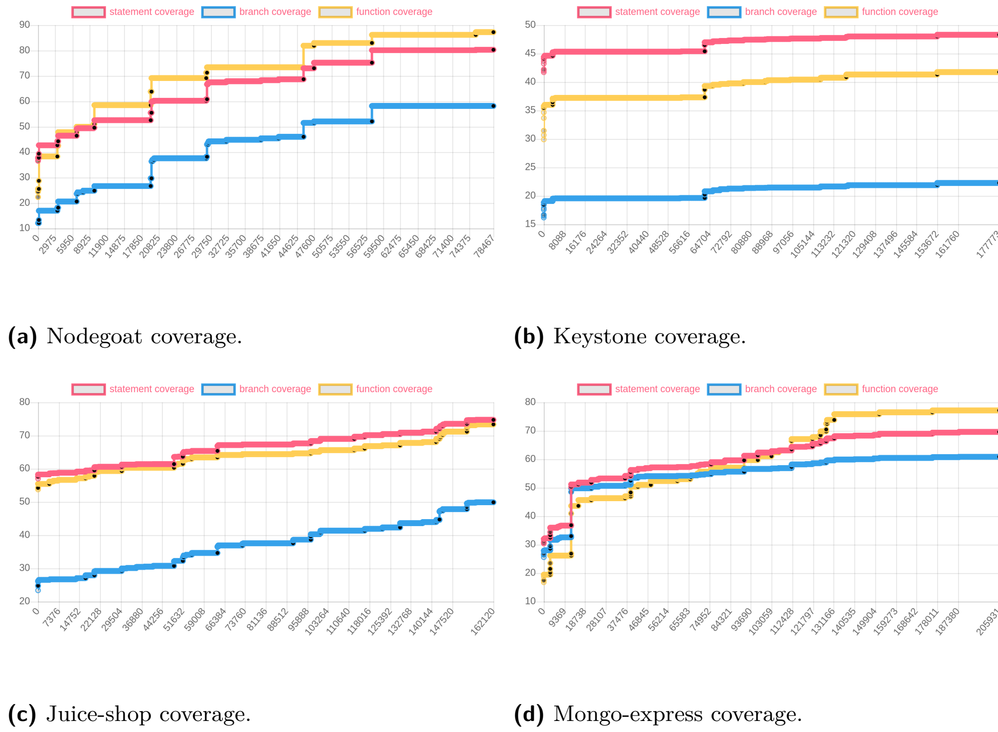
few existing Node.js benchmarks contain libraries only, and are unsuitable for evaluating web application fuzzers. For this reason, we created our own benchmark and open-sourced our evaluation framework to help with reproducibility. Trial-wise, contrary to most fuzzers, we tuned BackREST to be *deterministic*, meaning that every run of the fuzzer produces the same results and that a single trial per experiment is sufficient. Of course, for this to hold, we also assume that applications behave deterministically. To this end, we reset the state of applications after every run, and limit the number of concurrent requests to one. Finally, to ensure a fair comparison of BackREST against other state-of-the-art fuzzers, a co-author of this paper, who did not contribute nor had access to BackREST, was mandated to experiment with and tune the fuzzers to detect a maximum number of vulnerabilities in our benchmark applications.

5.2 API inference

Table 2 shows the number of endpoints and request parameters (excluding path parameters) that were inferred for each benchmark application. The numbers in parenthesis represent the number of additional endpoints and parameters that were identified using static type inference. Results clearly show that our crawler is very efficient at identifying the endpoints of an application while static type inference provides additional request parameters to fuzz.

5.3 Feedback-driven fuzzing

Table 3 compares the total coverage and runtime achieved by enabling the coverage feedback loop only (column C) and combined with taint feedback (column CT) against the baseline blackbox fuzzer (column B). Table 3 also lists speedups for coverage and taint feedback loops compared to baseline. Coverage-wise, enabling the coverage feedback loop, which skips payloads of a given type after $\mathcal{T}$ requests that did not increase coverage, achieves



(a) Nodegoat coverage.

(b) Keystone coverage.

(c) Juice-shop coverage.

(d) Mongo-express coverage.

Figure 4 Cumulative statement, branch, and function coverage (y axis) in function of the number of requests (x axis) for Nodegoat (a), Keystone (b), Juice-shop (c), and Mongo-express (d) with the baseline blackbox fuzzer.

approximately the same coverage (i.e. $\pm 2\%$) in a much faster way (i.e. speedup between 7.0$\times$ and 10.9$\times$). The slight variations in coverage can be explained by many different factors, such as the number of dropped requests, differences in scheduling and number of asynchronous computations, and differences in the application internal state. Indeed, the process of fuzzing puts the application under such a heavy load that exceptional behaviours become more common. Adding taint feedback on top of coverage feedback further decreases runtime, with speedups between 7.4$\times$ and 25.9$\times$. The slightly lower coverage can be explained by the fact that taint feedback forces the fuzzer to skip entire payload types, resulting in lower input diversity and slightly lower total coverage. Finally, the size of the API model for Apostrophe and the load that resulted from using the baseline fuzzer rendered the application unresponsive, and we killed the fuzzing session after 72 hours. Enabling taint feedback for Apostrophe almost halved the runtime compared to coverage feedback alone.

Figure 4 shows the cumulative coverage achieved by the baseline blackbox fuzzer on all applications but Apostrophe. For all applications, cumulative coverage evolves in a step-wise fashion (e.g. marked increases, followed by plateaus) where steps correspond to the fuzzer switching to a new endpoint. The plateaus that follow correspond to the fuzzer looping through its payload dictionary. These results support our coverage feedback heuristic, which is based on the assumption that the likelihood of covering new code by fuzzing a given endpoint decreases with the number of requests.

Table 4 Impact of the coverage (C) and taint (T) feedback loops on bug reports, compared to baseline (B).

Benchmark	(No)SQLi			Cmd injection			XSS			DoS		
	B	C	CT	B	C	CT	B	C	CT	B	C	CT
Nodegoat	0	0	3	0	0	3	5	5	5	0	0	0
Keystone	0	0	0	0	0	0	1	1	0	0	0	0
Apostrophe	0	0	0	0	0	0	1	1	1	2	1	1
Juice-Shop	1	1	2	0	0	1	4	1	1	1	0	0
Mongo-express	0	0	5	0	0	2	0	0	0	3	3	3
Total	1	1	10	0	0	6	11	8	7	6	4	4

5.4 A note on server-side state modelling

Many studies have shown that state-aware crawling of the client-side yields better coverage [73, 20, 74, 79, 32], and our crawler is no exception. Very little is known, however, about the impact of state-aware fuzzing of the server-side. To our knowledge, RESTler [21] is the first study to investigate stateful fuzzing of web services. While the authors have found a positive correlation between stateful fuzzing and increases in coverage, we have not observed a similar effect on our benchmark applications. Similar to RESTler, we attempted to model the state of our benchmark applications by inferring dependencies between endpoints. Specifically, we used the approach from [24] to infer endpoint dependencies from crawling logs and then constrained the fuzzing schedule of BackREST to honour them. This did not improve coverage for all but the Mongo-express application (data not shown). In this particular case, manual inspection revealed that the inferred dependencies were quite intuitive (e.g. insert a document before deleting it) and easily configured.

5.5 Vulnerability detection

Table 4 shows the number of unique true positive bug reports with the baseline fuzzer (column B), with coverage feedback (column C), and further adding taint feedback (column CT). We manually reviewed all reported vulnerabilities and identified the root causes. Table 4 does not list false positives for space and readability reasons. The only false positives were an XSS in Nodegoat that was reported by all three variations, and an SQLi in Keystone that was reported with taint feedback only. Also note that Table 4 lists three types of vulnerabilities (SQLi, command injection, and XSS) and one type of attack (DoS). We opted to list DoS for readability reasons. Indeed, the root causes of DoS are highly diverse (out-of-memory, infinite loops, uncaught exception, etc.) making it difficult to list them all. From our experience, the most prominent root cause for DoS in Node.js are uncaught exceptions. Indeed, contrary to many web servers, the Node.js front-end that listens to incoming requests is single-threaded. Crashing the front-end thread with an uncaught exception thus crashes the entire Node.js process [76].

Interestingly, enabling coverage feedback has no impact on the detection of SQLi and command injection vulnerabilities, suggesting that this optimisation could be enabled at no cost. Enhancing the fuzzer with taint feedback, however, consistently detects as many or more SQLi and command injection vulnerabilities. This is explained by the fact that taint inference does not rely on client-observable side-effects of a payload to detect vulnerabilities. This is especially obvious for command injection vulnerabilities, which are detected with

taint feedback only, for which observable side-effects are often hard to correlate to the root cause (e.g. slowdowns, internal server errors), compared to cross-site scripting, for example. Because the most critical injection flaws sit on the server-side of web applications, and are, by nature, harder to detect at the client side, the taint inference in BackREST gives it a tremendous edge over blackbox fuzzers. Table 4 also shows, however, that some cross-site scripting and denial-of-service vulnerabilities are missed when coverage and taint feedback are enabled. First, all missed XSS are stored XSS. Indeed, through sheer brute force, the baseline fuzzer manages to send very specific payloads that exploit stored XSS vulnerabilities and trigger side-effects that can be observed at the client (e.g. reflecting the payload in another page), while coverage and taint feedback loops caused these specific payloads to be skipped. To reliably detect stored XSS, taint analysis would need to track taint flows through storage, which implies shadowing every storage device (i.e. databases and file system) to store and propagate taint. This feature is beyond greybox fuzzing and is known to be tricky to implement, and costly from both time and memory perspectives [58]. Second, the missed denial-of-service vulnerabilities are due to: 1. a slow memory leak that requires several thousand requests to manifest in Apostrophe, and 2. a specific SQLite input that happened to be skipped with coverage and taint feedback.

Evaluating false negatives. In the context of a fuzzing session, false negatives are those vulnerabilities that are in the scope of a fuzzer, but that are missed. Accounting for false negatives requires an application with known vulnerabilities. The OWASP Nodegoat and Juice-Shop projects are deliberately vulnerable applications with seeded vulnerabilities. Both projects were built for educational purposes, and both have official solutions available, making it possible to evaluate false negatives. The solutions, however, list vulnerabilities from a penetration tester perspective; they list attack payloads, together with the error messages or screens they should lead to. For this reason, correlating the official solutions to BackREST reports is not trivial. For example, the Juice-Shop solution reports several possible different SQL and NoSQL injection attacks. From a fuzzing perspective, however, all these attacks share the same two root causes: calling specific MongoDB and Sequelize query methods with unsanitised inputs. In other words, while the official solutions report different exploits, BackREST groups them all under the same two vulnerability reports. For this study, we manually correlated all the SQLi, command injection, XSS, and DoS exploits in official solutions to vulnerabilities in the applications and found that BackREST reports them all, achieving a recall of 100% for Nodegoat and Juice-Shop.

5.6 Comparison with state-of-the-art

In this section, we compare BackREST against the arachni [1], w3af [7] and OWASP Zap [4] blackbox web application fuzzers. While we initially planned to also evaluate jÄk [79], our attempts at running it on our benchmark applications ultimately failed because of outdated dependencies (the code is 6 years old), authentication issues, and internal errors. To minimise bias and ensure a fair evaluation, all three remaining fuzzers were evaluated by a co-author of this paper who did not contribute and did not have access to BackREST, and was mandated to tune them to report a maximum number of vulnerabilities. All fuzzers were configured to scan for (No)SQLi, command injection, XSS, and DoS vulnerabilities. Significant care was also taken to configure all the fuzzers to authenticate into the applications and not log themselves out during a scan. Finally, after we discovered that the crawlers in arachni and w3af are fairly limited when it comes to navigating single-page web applications that

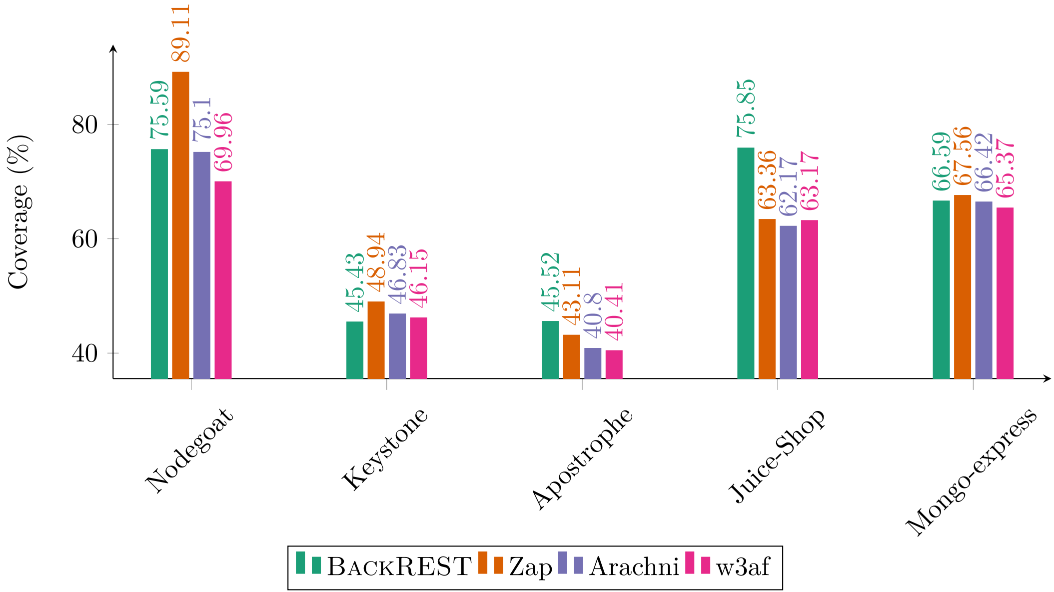


Figure 5 Coverage achieved by different fuzzers.

heavily rely on client-side JavaScript, we evaluated these fuzzers with seed URLs from a Zap crawling session. Zap internally uses the Crawljax [73] crawler that is better suited to navigate modern JavaScript-heavy applications.

Figure 5 shows the coverage that was achieved by the different fuzzers on the benchmark applications. BackREST consistently achieves comparable coverage to other fuzzers. Table 6 compares the number of bugs found by each fuzzer. For BackREST, we report the bugs found with coverage and taint feedback only. Apart from denial-of-service, BackREST consistently detects more vulnerabilities than other fuzzers, which we mostly attribute to the taint feedback loop. Indeed, while blackbox fuzzers can only observe the *side-effects* of their attacks through error codes and client-side inspection, BackREST can determine with high precision if a payload reached a sensitive sink and report vulnerabilities that would otherwise be difficult to detect in a purely blackbox fashion. Furthermore, we confirmed through manual inspection that apart from DoS, BackREST always reports a strict superset of the vulnerabilities reported by the other fuzzers. Deeper inspection revealed that the additional DoS found by Zap in Mongo-Express was due to a missing URL-encoded null byte payload (`%00`) in our payload dictionary.

It is very difficult to compare the performance of different web fuzzers, given the number of tunable parameters that each of them offers. For this reason, we focused our efforts on configuring them to maximise their detection power and did let them run until completion. The runtime of BackREST, as reported in Table 3, is directly proportional to the size of the API, which explains the longer runtime on Apostrophe. OWASP ZAP, our closest contender in terms of detected vulnerabilities, took between three minutes and three hours to complete a scan. Arachni took between ten minutes and one hour and a half, and w3af took between one and thirty minutes. Note that the low runtime of w3af is due to the fact that it stops its scan early if the application starts sending too many error responses.

Table 5 0-day vulnerabilities found by BackREST (B), Zap (Z), Arachni (A) and w3af (w3).

Codebase	Vulnerability	Found by	Taint only	Severity	Ref.
MarsDB	Command injection	B	✓	Critical	[39]
Sequelize	Denial-of-Service	B		Moderate	[38]
Apostrophe	Denial-of-Service	B		—	[89]
Apostrophe	Denial-of-Service	B		Low	[40]
Mongo-express[1)]	Command injection	B	✓	Critical	[62]
Mongo-express	Denial-of-Service	B, Z, A, w3		Medium	[71]
Mongodb-query-parser	Command injection	B	✓	Critical	[92]
MongoDB	Denial-of-service	B, Z		High	[49]

1) BackREST independently and concurrently found the same vulnerability

5.7 Reported 0-days

Table 5 lists the 0-days that were identified in benchmark applications and the fuzzers that reported them. The *taint only* column shows whether a particular 0-day was reported with taint feedback *only*. Out of all the vulnerabilities reported in Table 6, nine translated into 0-days, out of which six were reported by BackREST only. Several reasons explain why not all vulnerabilities translated to 0-day. First, recall that Nodegoat and Juice-Shop are deliberately insecure applications with seeded vulnerabilities. While BackREST detected several of them, they are not 0-days. Interestingly, however, BackREST did report non-seeded vulnerabilities in MarsDB, and Sequelize, which happen to be dependencies of Juice-Shop. Through the fuzzing of Juice-Shop, BackREST indeed triggered a command injection vulnerability in MarsDB and a denial-of-service in Sequelize. Second, the XSS that were reported in Apostrophe and Keystone are exploitable only in cases where the JSON response containing the XSS payload is processed and rendered in HTML. While we argue that returning JSON objects containing XSS payloads is a dangerous practice because it puts the consumers of the returned JSON object at risk, developers decided otherwise and did not accept our reports as vulnerabilities. Third, Mongo-express is a database management console; it deliberately lets its users inject arbitrary content. Hence, NoSQLi in Mongo-express can be considered as a *feature*. Otherwise, the command injection and denial-of-service vulnerabilities in Mongo-express and its dependencies all translated into 0-days, and so did the denials-of-services in Apostrophe.

For readers who might not be familiar with the Node.js ecosystem, it is important to underline how the MongoDB and Sequelize libraries are core to *millions* of Node.js applications. At the time of writing, MongoDB[4] had 1 671 653 *weekly* downloads while Sequelize[5] had 648 745. By any standard, these libraries are extremely heavily used, and well exercised.

[4] https://www.npmjs.com/package/mongodb
[5] https://www.npmjs.com/package/sequelize

Table 6 Number of vulnerabilities found by BackREST (B), Zap (Z), Arachni (A) and w3af (w3).

Benchmark	(No)SQLi				Cmd inj.				XSS				DoS			
	B	Z	A	w3	B	Z	A	w3	B	Z	A	w3	B	Z	A	w3
Nodegoat	3	3	0	2	3	0	0	3	5	4	2	3	0	0	0	0
Keystone	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Apostrophe	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
Juice-Shop	2	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0
Mongo-express	5	0	0	0	2	0	0	0	0	0	0	0	3	4	3	3
Total	**10**	4	1	3	**6**	0	0	3	**7**	5	2	3	**4**	**4**	3	3

Listing 5 Command injection vulnerability in MarsDB.

```
1 //Juice-Shop code
2 //Implements the /rest/track-order/{id} route
3 db.orders.find({ $where: "this.orderId === '" + req.params.id + "'" }).then(
4   order => { ... },
5   err => { ... }
6 );
7
8 //MarsDB code
9 $where: function(selectorValue, matcher) {
10   matcher._recordPathUsed('');
11   matcher._hasWhere = true;
12   if (!(selectorValue instanceof Function)) {
13     selectorValue = Function('obj', 'return ' + selectorValue);
14   }
15   return function(doc) {
16     return {result: selectorValue.call(doc, doc)};
17   }
18 };
```

6 Case studies

In this section, we detail some of the 0-days we reported in Table 5. We also explain some JavaScript constructs that might be puzzling to readers who are not familiar with the language. All the information presented in the following case studies is publicly available and a fix has been released for all but one of the vulnerabilities we present (MarsDB). In this particular case, the vulnerability report has been public since Nov 5th, 2019.

6.1 MarsDB command injection

MarsDB is an in-memory database that implements the MongoDB API. Listing 5 shows the command injection vulnerability in the MarsDB library that BackREST uncovered. Attacker-controlled input is injected in the client application at line 3, through a request parameter (bolded). The client application then uses the unsanitised tainted input to build a MarsDB `find` query. In Node.js, long-running operations, such as querying a database, are executed asynchronously. In this example, calling the `find` method returns a JavaScript *promise* that will be resolved asynchronously. Calling the `then` method of a promise allows to register handler functions for cases where the promise is fulfilled (line 4) or rejected (line 6). The query eventually reaches the `where` function of the MarsDB library at line 9 as the `selectorValue` argument. That argument is then used at line 13 to dynamically create a new function from a string. From a security perspective, calling the `Function` constructor

in JavaScript is roughly equivalent to calling the infamous `eval`; it dynamically creates a function from code supplied as a string. The newly created function is then called at line 16, which triggers the command injection vulnerability [39]. In this particular case, unless the payload is specifically crafted to: 1. generate a string that is valid JavaScript code, and 2. induce a side-effect that is observable from the client, it can be very difficult to detect this vulnerability in a purely blackbox manner. Thanks to taint feedback, BackREST can detect the command injection as soon as *any* unsanitised input reaches the `Function` constructor at line 16.

6.2 Sequelize DoS

Sequelize is a Node.js Object-Relational Mapper (ORM) for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. The code in Listing 6 is vulnerable to a DoS attack that crashes the Node.js server with an uncaught exception. First, an attacker-controllable SQL query is passed as the `sql` argument to the `run` function that executes SQL queries at line 1. The tainted query is assigned to various variables, the query is executed, and after its execution is eventually searched for the string "`sqlite_master`" at line 7, which is a special table in SQLite. If the search is successful, the query is then searched for the string "`SELECT sql FROM sqlite_master WHERE tbl_name`". If this search is unsuccessful and the query returned a `results` object that is not `undefined` (line 14), the `map` [6] method is called on the `results` object at line 16. This is where the DoS vulnerability lies. If the `results` object is not an array, it likely won't have a `map` method in its prototype chain, which will throw an uncaught `TypeError` and crash the Node.js process [38]. In summary, any request that includes the string "`sqlite_master`", but not "`SELECT sql FROM sqlite_master WHERE tbl_name`", and that returns a single value (i.e. not an array), will crash the underlying Node.js process.

Listing 6 Denial-of-Service (DoS) vulnerability in `Sequelize`.

```
1  run(sql, parameters) {
2    this.sql = sql;
3    const query = this;
4    function afterExecute(err, results) {
5      ...
6      if (query.sql.indexOf('sqlite_master') !== -1) {
7        if (query.sql.indexOf('SELECT sql FROM sqlite_master WHERE tbl_name') !== -1) {
8          result = results;
9          if (result && result[0] && result[0].sql.indexOf('CONSTRAINT') !== -1) {
10           result = query.parseConstraintsFromSql(results[0].sql);
11         }
12       }
13       else if (results !== undefined) {
14         // Throws a TypeError if results is not an array.
15         result = results.map(resultSet => resultSet.name);
16       }
17       else {
18         result = {}
19       }
20     }
21   }
22 }
```

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Listing 7 Denial-of-Service (DoS) vulnerability in `Apostrophe`.

```
1 self.routes.list = function(req, res) {
2   if (req.body.format === 'managePage') {
3     ...
4   } else if (req.body.format === 'allIds') {
5     ...
6   }
7   return self.listResponse(req, res, err, results);
8 };
```

6.3 Apostrophe DoS

Apostrophe is an enterprise content management system (CMS). Listing 7 shows a snippet of Apostrophe code that is vulnerable to a DoS attack. This code reads the `format` parameter of the request body, and checks if it is equal to "`managePage`" or "`allIds`", but misses a fallback option for cases where it is equal to neither. If this situation occurs, an uncaught exception is thrown, crashing the server [89].

6.4 Mongo-express command injections

Mongo-express is a MongoDB database management console. In versions prior to 0.54.0, it was calling an eval-like method with attacker-controllable input, leading to a command injection vulnerability. While BackREST independently detected this vulnerability, it was concurrently reported days before our own disclosure [62]. Interestingly, BackREST then revealed how the fix still enabled command injection. Indeed, the fix was to use the `mongo-db-query-parser` library to parse attacker-controlled input. The issue is that the library is using `eval` itself. Thanks to taint feedback, BackREST detected that tainted input was *still* flowing to an `eval` call, which we disclosed [92].

6.5 MongoDB DoS

Listing 8 Denial-of-Service (DoS) vulnerability in `MongoDB`.

```
1 function createCollection(db, name, options, callback) {
2   ...
3   executeCommand(db, cmd, finalOptions,
4     err => {
5       if (err) return handleCallback(callback, err);
6       handleCallback(
7         callback,
8         null,
9         // Throws an uncaught MongoError if the name argument is invalid
10        new Collection(db, db.s.topology, db.s.databaseName, name,
11            db.s.pkFactory, options)
12      );
13    }
14  );
15  ...
16 }
```

MongoDB is a document-based NoSQL database with drivers in several languages. Listing 8 shows a snippet from the MongoDB driver that has a DoS vulnerability. This code gets executed when new collections are created in a MongoDB database. If the name

of the collection to be created is attacker-controllable, and the attacker supplies an invalid collection name, the call to the `Collection` constructor at line 10 fails and throws an uncaught `MongoError` that crashes the Node.js process [49]. The taint feedback loop quickly reports that a tainted collection name flows to the `Collection` constructor, enabling BackREST to trigger the vulnerability faster.

7 Related Work

Web application modelling. Modelling for web applications has a long and rich history in the software engineering and testing communities. Modelling methods broadly fall into three main categories: graph-based, UML-based, and FSM-based. Graph-based approaches focus on extracting navigational graphs from web applications and applying graph-based algorithms (e.g. strongly connected components, dominators) to gain a better understanding of the application [102, 31]. UML-based approaches further capture the interactions and flows of data between the different components of a web application (e.g. web pages, databases, and server) as a UML model [87, 102, 19]. To facilitate automated test case generation, FSM-based approaches instead cluster an application into sub-systems, model each with a finite-state machine and unify them into a hierarchy of FSMs [18]. All these approaches were designed to model *stateful* web applications, which were the norm back in the early 2000s. Since then, web development practices evolved, developers realised that building server-side applications that are as *stateless* as possible improves maintainability, and the REST protocol, which encourages statelessness, gained significant popularity.

Grammar inference. Automated learning of grammars from inputs is a complementary and very promising research area [44, 54, 22]. To efficiently learn a grammar from inputs, however, current approaches either require: 1. very large datasets of input to learn from ([44]); 2. highly-structured parser code that reflects the structure of the underlying grammar ([54]); or 3. a reliable oracle to determine whether a given input is well-formed ([22]). Unfortunately, very few web applications meet any of these criteria, making model inference the only viable alternative. Compared to synthesised grammars, REST models are also easier to interpret.

Model-based fuzzing. Model-based fuzzing derives input from a user-supplied [82, 21], or inferred [90, 91, 48, 33, 32] model. Contrary to grammar-based fuzzing [52, 53, 43], input generation in model-based fuzzing is not constrained by a context-free grammar. While grammar-based fuzzers generally excel at fuzzing language parsers, model-based fuzzers are often better suited for higher-level programs with more weakly-structured input.

REST-based fuzzing. Most closely related to our work are REST-based fuzzers. In recent years, many HTTP fuzzers have been extended to support REST specifications [59, 85, 98, 83, 100], but received comparatively little attention from the academic community. RESTler [21] is the exception. It uses a user-supplied REST specification as a model for fuzzing REST-based services in a blackbox manner. To better handle stateful services, RESTler enriches its model with inferred dependencies between endpoints. As we highlighted in Subsection 5.4, we haven't found endpoint dependency inference to have a significant impact on the coverage of all but one of our benchmark applications. For Mongo-express, we found that the inferred dependencies were trivial, easily configurable and did not justify the added complexity.

Taint-based fuzzing. Taint-based fuzzing uses dynamic taint analysis to monitor the flow of attacker-controlled values in the program under test and to guide the fuzzer [37, 23, 67]. Wang et al. [103, 104] use taint analysis to identify the parts of an input that are compared against checksums. Similarly, Haller et al. use taint analysis to identify inputs that can trigger buffer overflow vulnerabilities [47], while Vuzzer [86] uses it to identify parts of an input that are compared against magic bytes or that trigger error code. Taint analysis has also been used to map input to the parser code that processes it and to infer an input grammar [53]. More closely related to our work, the KameleonFuzz tool [33] uses taint inference on the client-side of web applications to detect reflected inputs and exploit cross-site scripting vulnerabilities. BlackWidow [34] implements several stateful crawling strategies in combination with client-side taint inference to detect stored and reflected cross-site scripting vulnerabilities in multi-page PHP applications. To our knowledge, BackREST is the first web application fuzzer to implement a *server-side* taint inference feedback loop.

Web application security scanning. The process of exercising an application with automatically generated malformed, or malicious input, which is nowadays known as fuzzing, is also known as security scanning, attack generation, or vulnerability testing in the web community. Whitebox security scanning tools include the QED tool [72] that uses goal-directed model checking to generate SQLi and XSS attacks for Java web applications. The seminal Ardilla paper [58] presented a whitebox technique to generate SQLi and XSS attacks using taint analysis, symbolic databases, and an attack pattern library. Ardilla was implemented using a modified PHP interpreter, tying it to a specific version of the PHP runtime. Similarly, Wassermann et al. also modified a PHP interpreter to implement a concolic security testing approach [105]. BackREST instrumentation-based analyses decouples it from the runtime, making it easier to maintain over time. More recent work on PHP application security scanning includes Chainsaw [12] and NAVEX [13] that use static analysis to identify vulnerable paths to sinks and concolic testing to generate concrete exploits. Unfortunately, the highly dynamic nature of the JavaScript language makes any kind of static or symbolic analysis extremely difficult. State-of-the-art static analysis approaches can now handle some libraries and small applications [95, 75, 60] but concolic testing engines still struggle to handle more than a thousand lines of code [30, 69, 15]. For this reason, blackbox scanners like OWASP Zap [4], Arachni [1] or w3af [7], which consist of a crawler coupled with a fuzzing component, were the only viable option for security scanning of Node.js web applications. With BackREST, we showed that lightweight coverage and taint inference analyses are well-suited to dynamic languages for which static analysis is still extremely challenging.

Web vulnerability detection and prevention. In the past two decades, a very large body of work has focused on detecting and preventing vulnerabilities in web applications. The seminal paper by Huang et al. introduced the WebSSARI tool [55] that used static analysis to detect vulnerabilities and runtime protection to secure potentially vulnerable areas of a PHP application. In their 2005 paper, Livshits and Lam showed how static taint analysis could be used to detect injection vulnerabilities, such as SQLi and XSS, in Java web applications [68]. The Pixy tool [56] then showed how static taint analysis could be ported, to the PHP language to detect web vulnerabilities in PHP web applications. The AMNESIA tool [46] introduced the idea of modelling SQL queries with static analysis and checking them against the model at runtime. This idea was further formalised by Su et al. [96], applied to XSS detection [99] and is still used nowadays to counter injection attacks in Node.js applications [94].

JavaScript vulnerability detection. As Web 2.0 technologies gained in popularity, the client-side of web applications became richer, and researchers started investigating the JavaScript code that runs in our browsers. It became quickly obvious, however, that existing static analysis techniques could not be easily ported to JavaScript, and that dynamic techniques were better suited for highly dynamic JavaScript code [88]. Dynamic taint analysis thus started to gain popularity, and was particularly successful at detecting client-side DOM-based XSS vulnerabilities [63, 78]. In the meantime, in 2009, the first release of Node.js, which brings JavaScript to the server-side, came out, and is now powering millions of web applications worldwide. Despite its popularity, however, the Node.js platform comparatively received little attention from the security community [76, 81, 29], with only two studies addressing injection vulnerabilities [94, 75].

8 Conclusion

We presented BackREST, the first fully automated model-based, coverage- and taint-driven greybox fuzzer for web applications. BackREST guides a state-aware crawler to automatically infer REST-like APIs, and uses coverage feedback to avoid fuzzing thoroughly covered code. BackREST makes a novel use of taint feedback to focus the fuzzing session on likely vulnerable areas, guide payload generation, and detect more vulnerabilities. Compared to a baseline version without taint and coverage feedback, BackREST achieved speedups ranging from 7.4× to 25.9×. BackREST also consistently detected more (No)SQLi, command injection, and XSS vulnerabilities than three state-of-the-art web fuzzers and detected six 0-days that were missed by all other fuzzers.

Depending on the context in which fuzzing is used, aspects like runtime, or number, depth, or severity of bugs reported will be prioritised. In our industrial setting, where fuzzing is used as a nightly security testing tool, time is of essence. By extending a blackbox web application fuzzer with coverage and taint feedback loops that helps it *skip* and *select* inputs, we showed how it can detect *more* vulnerabilities *faster*. In our setting, the initial investment in development time was quickly absorbed by the time saved during each fuzzing session, without accounting for the additional bugs found. The analyses we described are simple enough to be applied to a vast number of existing black-box web application fuzzers and we hope that our study will trigger further research in this area.

References

1. Arachni. URL: `https://www.arachni-scanner.com/`.
2. Burp suite. URL: `https://portswigger.net/burp`.
3. Empirical Evaluation Guidelines. URL: `https://www.sigplan.org/Resources/EmpiricalEvaluation/`.
4. OWASP Zed Attack Proxy. URL: `https://www.zaproxy.org/`.
5. Peach fuzzer community edition. URL: `https://www.peach.tech/resources/peachcommunity/`.
6. A python client for swagger enabled rest api. URL: `https://github.com/pyopenapi/pyswagger`.
7. w3af. URL: `http://w3af.org/`.
8. AngularJS. `https://angularjs.org/`, 2021. Accessed: 2021-02-1.
9. React.js. `https://reactjs.org/`, 2021. Accessed: 2021-02-1.
10. Stack Overflow Developer Survey. `https://insights.stackoverflow.com/survey/2020#technology-web-frameworks`, 2021. Accessed: 2021-02-1.

11 The ElementTree XML library. `https://docs.python.org/3/library/xml.etree.elementtree.html`, 2021. Accessed: 2021-03-24.

12 Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652, 2016.

13 Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

14 Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 13–18. ACM, 2015.

15 Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Constraint programming for dynamic symbolic execution of javascript. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 1–19. Springer, 2019.

16 Pedram Amini, Aaron Portnoy, and Ryan Sears. Sulley. `https://github.com/OpenRCE/sulley`.

17 Krishnan Anantheswaran, Corey Farrell, and contributors. Istanbul: Javascript test coverage made simple. URL: `https://istanbul.js.org/`.

18 Anneliese A Andrews, Jeff Offutt, and Roger T Alexander. Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4(3):326–345, 2005.

19 Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara. Understanding web applications through dynamic analysis. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 120–129. IEEE, 2004.

20 Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of JavaScript Web Applications. In *ICSE*, 2011.

21 Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

22 Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.

23 Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825. IEEE, 2012.

24 Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277, 2011.

25 Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

26 Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

27 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

28 Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 5.1]. URL: `https://mitmproxy.org/`.

29 James Davis, Arun Thekumparampil, and Dongyoon Lee. Node. fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 145–160, 2017.

30 Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. Type-aware concolic testing of javascript programs. In *Proceedings of the 38th International Conference on Software Engineering*, pages 168–179, 2016.

31 Eugenio Di Sciascio, Francesco M Donini, Marina Mongiello, and Giacomo Piscitelli. Anweb: a system for automatic support to web application verification. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 609–616, 2002.

32 Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 523–538, 2012.

33 Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.

34 Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142, 2021.

35 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

36 Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

37 Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

38 Françcois Gauthier. Denial of Service – sequelize. `https://www.npmjs.com/advisories/1142`.

39 François Gauthier. Command Injection – marsdb. `https://www.npmjs.com/advisories/1122`.

40 François Gauthier. Denial of Service – apostrophe. `https://www.npmjs.com/advisories/1183`.

41 François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. Affogato: Runtime Detection of Injection Attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 94–99. ACM, 2018.

42 Patrice Godefroid. Fuzzing: hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

43 Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

44 Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

45 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices*, 47(6):239–250, 2012.

46 William GJ Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, 2005.

47 Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 49–64, 2013.

48 HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

49 Behnaz Hassanshahi. Denial of Service – mongodb. `https://www.npmjs.com/advisories/1203`.

50 Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *29th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

51 Marijn Haverbeke. A JavaScript code analyzer for deep, cross-editor language support. `https://ternjs.net/`. Accessed: 17-06-2019.

52 Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012.

53 Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725. IEEE, 2016.

54 Matthias Höschele and Andreas Zeller. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 31–34. IEEE Press, 2017.

55 Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

56 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

57 Rody Kersten, Kasper Søe Luckow, and Corina S. Pasareanu. POSTER: afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2511–2513. ACM, 2017.

58 Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st international conference on software engineering*, pages 199–209. IEEE, 2009.

59 KissPeter. APIFuzzer. `https://github.com/KissPeter/APIFuzzer`. Accessed: 04-07-2019.

60 Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for javascript object-manipulating programs. *Software: Practice and Experience*, 49(5):840–884, 2019.

61 Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for JavaScript library analysis. In *Proceedings of the 41st International Conference on Software Engineering*, pages 83–93. IEEE Press, 2019.

62 Jonathan Leitschuh. Remote Code Execution – mongo-express. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10758`.

63 Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

64 Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

65 Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

66 Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

67 Guangcheng Liang, Lejian Liao, Xin Xu, Jianguang Du, Guoqiang Li, and Henglong Zhao. Effective fuzzing based on dynamic taint analysis. In *2013 Ninth International Conference on Computational Intelligence and Security*, pages 615–619. IEEE, 2013.

68 V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

69 Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 196–199, 2017.

70 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

71 Trong Nhan Mai. Denial of Service (DoS) – mongo-express. `https://snyk.io/vuln/SNYK-JS-MONGOEXPRESS-1085403`.

72 Michael C Martin and Monica S Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *USENIX Security symposium*, pages 31–44, 2008.

73 Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE, 2008.

74 Amin Milani Fard and Ali Mesbah. Feedback-directed exploration of web applications to derive test models. In *ISSRE*, 2013.

75 Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of node. js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 455–465, 2019.

76 Andres Ojamaa and Karl Düüna. Assessing the security of node. js platform. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 348–355. IEEE, 2012.

77 Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: coverage-guided property-based testing in java. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 398–401. ACM, 2019.

78 Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Dexterjs: robust testing platform for dom-based xss vulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 946–949, 2015.

79 Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *RAID*, 2015.

80 Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168, 2017.

81 Brian Pfretzschner and Lotfi ben Othmane. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–6, 2017.

82 Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 543–553, 2016.

83 Qualys. Web application scanning. `https://www.qualys.com/apps/web-app-scanning/`. Accessed: 04-07-2019.

84 Sazzadur Rahaman, Gang Wang, and Danfeng Yao. Security certification in payment card industry: Testbeds, measurements, and recommendations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 481–498, 2019.

85 Rapid7. Swagger Utility. `https://appspider.help.rapid7.com/docs/swagger-utility`. Accessed: 04-07-2019.
86 Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.
87 Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 25–34. IEEE, 2001.
88 Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.
89 Max Schlüter. Server crash on POST request. `https://github.com/apostrophecms/apostrophe/issues/1683`.
90 Martin Schneider, Jürgen Großmann, Ina Schieferdecker, and Andrej Pietschker. Online model-based behavioral fuzzing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 469–475. IEEE, 2013.
91 Martin Schneider, Jürgen Großmann, Nikolay Tcholtchev, Ina Schieferdecker, and Andrej Pietschker. Behavioral fuzzing operators for uml sequence diagrams. In *International Workshop on System Analysis and Modeling*, pages 88–104. Springer, 2012.
92 Ben Selwyn-Smith. Remote Code Execution – mongodb-query-parser. `https://www.npmjs.com/advisories/1448`.
93 SmartBear. Openapi specification (fka swagger restful api documentation specification). `https://swagger.io/specification/v2/`. Accessed: 04-07-2019.
94 Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *NDSS*, 2018.
95 Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.
96 Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *Acm Sigplan Notices*, 41(1):372–382, 2006.
97 Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.
98 TeeBytes. TnT-Fuzzer. `https://github.com/Teebytes/TnT-Fuzzer`. Accessed: 04-07-2019.
99 Mike Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *2009 30th IEEE symposium on security and privacy*, pages 331–346. IEEE, 2009.
100 Alexandre Teyar. Swurg. `https://github.com/portswigger/openapi-parser`. Accessed04-07-2019.
101 John Toman and Dan Grossman. Concerto: a framework for combined concrete and abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3(POPL):43, 2019.
102 Paolo Tonella and Filippo Ricca. Dynamic model extraction and statistical analysis of web applications. In *Proceedings. Fourth International Workshop on Web Site Evolution*, pages 43–52. IEEE, 2002.
103 Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.
104 Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):1–28, 2011.
105 Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, 2008.

106 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

107 Michal Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2015.

Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis

Dongjie He ✉
The University of New South Wales, Sydney, Australia

Jingbo Lu ✉
The University of New South Wales, Sydney, Australia

Jingling Xue ✉
The University of New South Wales, Sydney, Australia

Abstract

Existing whole-program context-sensitive pointer analysis frameworks for Java, which were open-sourced over one decade ago, were designed and implemented to support only method-level context-sensitivity (where all the variables/objects in a method are qualified by a common context abstraction representing a context under which the method is analyzed). We introduce QILIN as a generalized (modern) alternative, which has been open-sourced on GitHub, to support the current research trend on exploring **fine-grained context-sensitivity** (including variable-level context-sensitivity where different variables/objects in a method can be analyzed under different context abstractions at the variable level), **precisely**, **efficiently**, and **modularly**. To meet these **four** design goals, QILIN is developed as an imperative framework (implemented in Java) consisting of a fine-grained pointer analysis kernel with parameterized context-sensitivity that supports on-the-fly call graph construction and exception analysis, solved iteratively based on a new carefully-crafted incremental worklist-based constraint solver, on top of its handlers for complex Java features.

We have evaluated QILIN extensively using a set of 12 representative Java programs (popularly used in the literature). For method-level context-sensitive analyses, we compare QILIN with DOOP (a declarative framework that defines the state-of-the-art), QILIN yields logically the same precision but more efficiently (e.g., 2.4x faster for four typical baselines considered, on average). For fine-grained context-sensitive analyses (which are not currently supported by open-source Java pointer analysis frameworks such as DOOP), we show that QILIN allows seven recent approaches to be instantiated effectively in our parameterized framework, requiring additionally only an average of 50 LOC each.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Pointer Analysis, Fine-Grained Context Sensitivity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.30

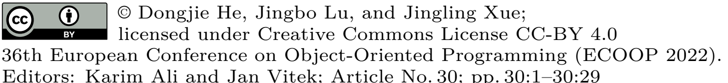
Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.6

Funding Supported by ARC Grants DP180104069 and DP210102409.

Acknowledgements We thank all the reviewers for their constructive comments.

1 Introduction

Pointer analysis, which approximates statically the possible run-time objects that may be pointed to by a variable in a program, is the basis of nearly all the other static program analyses. There are many significant applications, including call graph construction [23, 1, 38], program understanding [46, 36], bug detection [34, 55, 27, 10], security analysis [4, 11, 13], compiler optimization [9, 47], and symbolic execution [52, 21, 51].


© Dongjie He, Jingbo Lu, and Jingling Xue;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 30; pp. 30:1–30:29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For object-oriented languages, context-sensitive pointer analyses are the most common class of precise pointer analyses [48, 32, 23, 42, 44]. Existing (whole-program) pointer analysis frameworks for Java, such as Doop [8], Wala [16], Jchord [34], and Paddle [24], were all designed and implemented over a decade ago by supporting *method-level context-sensitivity* only. For these traditional frameworks, when a method is analyzed under a given context abstraction (e.g., a k-limited context string [41] with k being fixed for the program), all the variables/objects in the method are analyzed uniformly under that given context abstraction.

For the past decade, Doop has been the most widely used context-sensitive pointer analysis framework for Java [42, 22, 43, 48, 49, 26, 17, 18, 19]. Doop encodes a pointer analysis declaratively by using Datalog (a logic-based language) to define pointer-related relations (in terms of Datalog rules) and a Datalog engine to infer the points-to facts. Thus, its performance is largely determined by the Datalog engine used and can be also sensitive to both automatic and manual optimizations applied to the Datalog rules. For example, a recent Datalog engine porting effort for replacing LogicBlox with Soufflé [20, 39] has boosted its performance by up to 4x [3]. When it was released in 2009 [8], Doop was shown to outperform Paddle [24] (the then state-of-the-art framework with the core of its pointer analysis algorithm performed in Datalog declaratively but the rest coded in Java imperatively) significantly. In addition, Doop was then argued to cost less human effort in implementing different pointer analyses with different flavors of context-sensitivity as they can all be specified modularly as variations on a common code base.

Currently, one emerging research trend is shaping the future of research on context-sensitive pointer analyses. To analyze large programs more scalably with more flexible efficiency/precision trade-offs, context-sensitivity is becoming increasingly more fine-grained. Method-level context-sensitivity can now be selective [43, 19, 25] (with only a subset of methods in the program being analyzed context-sensitively) or partial [30, 14] (with only a subset of variables/objects in a method being analyzed context-sensitively). In the future, pointer analysis frameworks are expected to support *variable-level context-sensitivity*, which allows different variables/objects in a method to be analyzed under different context abstractions, in order to enlarge the space of efficiency/precision trade-offs made. As for the option of extending Doop to support such fine-grained context-sensitivity, we have made significant efforts, but its resulting performance can often be disappointing due to possibly poor join orders selected by its underlying Soufflé Datalog engine [39, 20] used. Understandably, while the authors of [43, 19, 25] implemented trivially their selective method-level context-sensitivity in Doop and observed the desired efficiency/precision trade-offs, the authors of [30, 14] had to settle with some in-house implementations of their partial method-level context-sensitivity in Soot [23] imperatively in order to achieve the expected efficiency/precision trade-offs.

We introduce Qilin, a modern framework (implemented imperatively in Java) for supporting Java pointer analyses with (1) **fine-grained context-sensitivity**, (2) **precisely**, (3) **efficiently**, and (4) **modularly**. How to support (1) subject to (2) – (4) is nontrivial both scientifically and engineering-wise and was not done before. For example, achieving (3) and (4) requires Qilin to adopt new scientific approaches to specify and conduct pointer analysis when different variables/objects in a method are analyzed under different context abstractions imperatively. Achieving (2) requires Qilin to handle the full semantic complexity of Java, involving huge engineering efforts. Given that Doop has been tuned for supporting method-level context-sensitivity for over a decade, can Qilin outperform Doop while achieving the same precision? In addition, can Qilin support a variety of fine-grained context-sensitivity well? Qilin addresses these challenges, making the following contributions:

- QILIN represents the first imperative Java pointer analysis framework for supporting fine-grained context-sensitivity, precisely, efficiently and modularly.
- QILIN achieves its efficiency and modularity by decoupling the analysis logic for a given analysis algorithm from its implementation in the following novel way:
  - QILIN includes a pointer analysis kernel (supporting both on-the-fly call graph construction and on-the-fly exception analysis) with parameterized fine-grained context-sensitivity, allowing different flavors (i.e., granularities) of fine-grained context-sensitivity to be specified (i.e., instantiated) modularly as variations on a common code base (even in the fully imperative setting).
  - QILIN includes a new incremental worklist-based constraint solver that has been generalized in a non-trivial manner from a traditional incremental worklist-based constraint solver for supporting context-insensitive pointer analyses [23]. As existing solvers are limited to method-level context-sensitivity, we have crafted our solver carefully in order to support fine-grained context-sensitive pointer analyses efficiently.
- QILIN covers the same complex Java features and semantic complexities (e.g., reflection, native code, threads, etc.) as DOOP, delivering the same analysis precision.
- QILIN anchors around it a tool suite consisting of not only all method-level context-sensitive analyses supported by DOOP but also a wide range of fine-grained analyses.
- QILIN is evaluated with a set of 12 representative Java benchmarks and applications (popularly used in the literature). For method-level context-sensitive analyses, QILIN (which is currently single-threaded) is 2.4x faster than DOOP (running with 8 threads under its best thread configuration) for four typical baselines considered on average while achieving exactly the same precision. Unlike DOOP, QILIN supports effectively fine-grained context-sensitive analyses, by enabling seven recent approaches to be instantiated in its parameterized framework with an average of 50 LOC being added only.

QILIN is designed to be an open-source project (released at `https://github.com/QiLinPTA/QiLin/`), consisting of currently about 20.3 KLOC in Java (including 4.7 KLOC only at its core for performing parameterized pointer analysis). As a highly-configurable pointer analysis framework, QILIN provides benefits for both researchers and end users. For researchers, QILIN can help them both experiment with new ideas more quickly than if they have to conduct their own in-house implementations of their pointer analysis algorithms as in [14, 28, 15] and evaluate their ideas by making an apples-to-apples comparison against the state of the art in the same framework. For end users, QILIN can help them build their client application tools, such as bug detectors and program verifiers, by choosing some existing configured pointer analyses that are best suited to their needs. We plan to grow and maintain this open-source pointer analysis framework on GitHub to provide a common framework for researchers and practitioners to design, implement and evaluate different analyses for Java programs.

The rest of this paper is organized as follows. Section 2 provides some background knowledge and motivates this work. Section 3 introduces our QILIN framework. In Section 4, we demonstrate how to create, i.e., instantiate a number of fine-grained context-sensitive pointer analyses modularly in QILIN. Section 5 provides an extensive evaluation of QILIN. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 Background and Motivation

We discuss context-sensitive pointer analyses by first reviewing method-level context-sensitivity and then motivating its generalization to variable-level context-sensitivity.

2.1 Method-Level Context-Sensitivity

Traditionally, context-sensitive approaches analyze a method separately under different calling contexts that abstract its different run-time invocations. Under such *method-level context-sensitivity*, whenever a method is analyzed for a given context, all its (local) variables and (allocated) objects are qualified by, i.e., analyzed under that context. For object-oriented languages, two common context abstractions are callsites (callsite-sensitivity [40]) and receiver objects, or precisely, their allocation sites (object-sensitivity [32, 33]). The two other variations are type-sensitivity [42] and hybrid sensitivity [22].

To tame the combinatorial explosion of contexts encountered in practice, a context is often represented by a k-limited context string, i.e., a sequence of k context elements [41]. Under k-limiting, two representative forms of context-sensitivity for object-oriented languages are: (1) *k-callsite-sensitivity* [40], which distinguishes the contexts of a method by its k-most-recent callsites, and (2) *k-object-sensitivity* [32, 33]), which distinguishes the contexts of a method by its receiver's k-most-recent allocation sites.

Despite k-limiting, the context explosion problem still occurs frequently in analyzing large programs [42, 48, 19], causing context-sensitive pointer analyses to be inefficient even when they are scalable (for usually only small values of k). Instead of applying k-limiting uniformly (with a fixed value of k) to all the methods (i.e., all the variables/objects) in the program, researchers have recently demonstrated that making context-sensitivity more fine-grained can lead to more flexible efficiency/precision trade-offs and better scalability. As a result, method-level context-sensitivity can now be selective [43, 19, 25] (with a subset of methods in the program being analyzed context-sensitively) and partial [30, 14] (with a subset of variables/objects in a method being analyzed context-sensitively).

2.2 Variable-Level Context-Sensitivity

In the future, variable-level context-sensitivity (that includes naturally method-level context-sensitivity as a special case) can be investigated by using Qilin. Under such *fine-grained context-sensitivity* in its full generality, different variables/objects in a method can be analyzed under different contexts (e.g., different k-limited context strings with different values of k). This will significantly enlarge the space of context-sensitive pointer analyses that researchers and practitioners can experiment with in order to achieve the most flexible efficiency/precision trade-offs and best scalability possible for their pointer analysis problems considered.

```
1 class A { Object f; }
2 class B {
3   Object foo(Object x, A a) {
4     a.f = x;
5     Object t = a.f;
6     System.out.print(t);
7     return x; }
8 }
```

```
9 void main() {
10   Object o1 = new Object(); // O1
11   B b1 = new B(); // B1
12   A a1 = new A(); // A1
13   Object v1 = b1.foo(o1, a1);
14   Object o2 = new Object(); // O2
15   B b2 = new B(); // B2
16   Object v2 = b2.foo(o2, a1);}
```

Figure 1 A motivating example program.

2.3 Example

Consider a simple program given in Figure 1, where `foo()` is called twice, once on receiver `B1` in line 13 and once on receiver `B2` in line 16. If the program is analyzed context-insensitively, these two calls will be conflated, and consequently, the parameter `x` will also be conflated, preventing the analysis from proving that the two calls actually return two distinct objects (i.e., `O1` and `O2`). As a result, `v1` and `v2` are concluded to point to both `O1` and `O2` conservatively.

However, a context-sensitive analysis that distinguishes the two calls will be able to infer that `v1` points to `O1` only and `v2` points to `O2` only. Without loss of generality, let us consider 1-object-sensitivity [32, 33], under which these two calls will be distinguished by its two different receiver objects, `B1` and `B2`, used. Thus, under method-level context-sensitivity, `foo()` is analyzed twice, with its four variables `this`, `x`, `a` and `t` being analyzed once under context [`B1`] and once under context [`B2`]. As the contexts of its parameter `x` are distinguished under the two calls, `v1` is found to point to `O1`, i.e., the object pointed to by `x` under [`B1`], and similarly, `v2` is found to point to `O2`, i.e., the object pointed to by `x` under [`B2`].

However, applying method-level context-sensitivity to `foo()` in this program is overkill. Under fine-grained context-sensitivity, we can conduct 1-object-sensitive analysis to `foo()` exactly as before except that we only need to distinguish its parameter `x` context-sensitively. Note that all the variables/objects in `main()` are naturally context-insensitive. By analyzing only `x` in this program context-sensitively, the resulting fine-grained analysis will be faster while losing no precision at all for all its variables/objects.

3 Designing the Qilin Framework

Figure 2 gives the architecture of Qilin, built on top of Soot [53], for supporting fine-grained context-sensitive pointer analysis for Java programs. Currently, Qilin's toolbox (①) includes not only all the method-level context-sensitive pointer analyses that are supported by Doop [8] but also a range of recently proposed representative fine-grained analyses, as will be elaborated in Section 4. In this section, we describe how Qilin is designed to support fine-grained context-sensitivity, precisely, efficiently and modularly in terms of its four major components depicted at ② – ⑤. In Section 3.1, we explain how we parameterize variable-level context-sensitivity to allow different flavors of context-sensitive analyses to be specified modularly (②). We then formalize our parameterized pointer analysis in Section 3.2 (③) and introduce our new incremental worklist-based constraint solver for solving it efficiently in Section 3.3 (④). In Section 3.4, we describe how Qilin covers complex Java features in order to support pointer analyses precisely for real-world Java programs (⑤).

It should be pointed out that in Qilin, all the pointer analyses (including what is provided in its toolbox (①)) are instantiated (concretized) in terms of ② as variations on a common code base consisting of ③ – ⑤, even though Qilin is implemented imperatively in Java.

3.1 Parameterized Context-Sensitivity

In Qilin, as depicted at ② in Figure 2, context-sensitivity for a given analysis is defined by a set of three parameters, a context constructor, a context selector, and a heap abstractor, each of which can be instantiated to support different flavors (i.e., granularities) of context-sensitivity from the method level to the variable level.

The *context constructor*, denoted `Cons`, is used to create the contexts required for analyzing a method in the traditional manner. Therefore, this parameter alone will be sufficient to specify different flavors of method-level context sensitivity considered traditionally, including

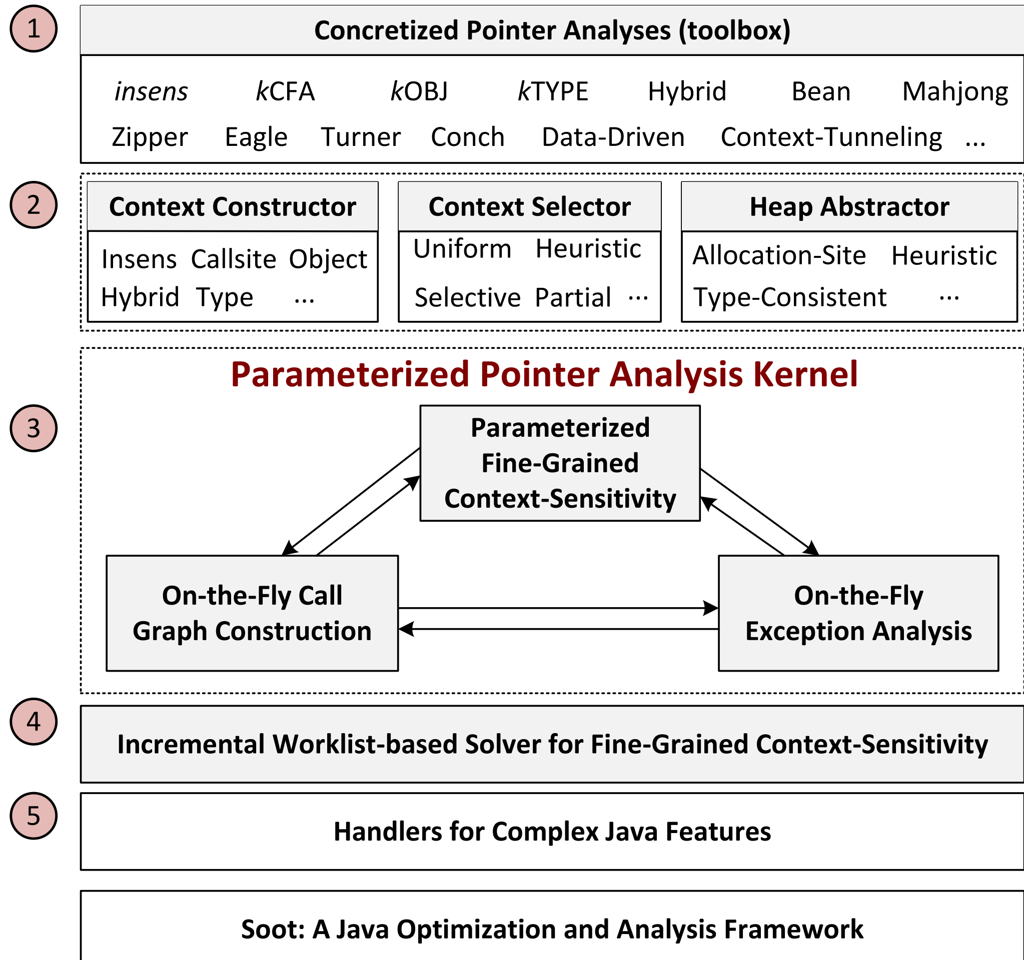


Figure 2 The architecture of Qilin.

"Insens", "CallSite", "Object", "Type", and "Hybrid", which will be instantiated in Section 4.1. The *context selector*, denoted `Sel`, is used to define the contexts required for analyzing variables/objects (based on the contexts of their containing methods specified by `Cons`) to support fine-grained context-sensitivity, including "Uniform", "Heuristic", "Selective", and "Partial", which will be instantiated in Section 4.2. The *heap abstractor*, denoted `HeapAbs`, is used to define an abstraction of the objects in the heap, including "Allocation Site", "Heuristic", and "Type-Consistent", which will be instantiated in Section 4.3.

As context-sensitivity is parameterized (②), an understanding about its actual instantiations is not needed now in order to understand its other three components (③ – ⑤), which will be described in turn below.

3.2 Parameterized Pointer Analysis

We describe our parameterized pointer analysis (③ in Figure 2) that supports on-the-fly call graph construction [23] and exception analysis [7] by considering a simplified subset of Java, with only eight kinds of labeled statements given in Table 1. Note that "x = **new** T(...)" in standard Java is modeled as "x = **new** T; x.`<init>`(...)", where `<init>`(...)

is the corresponding constructor invoked. The control flow statements are not considered because QILIN supports only context-sensitive analyses just as DOOP [8]. In our formalism (for simplicity and without loss of generality), every method is assumed to have one return statement "**return** *ret*", where *ret* is its *return variable*, and one special catch statement "**catch** *eret*" for catching all throwable objects thrown out of the method.

Table 1 Eight kinds of statements analyzed by QILIN.

Kind	Statement	Kind	Statement
NEW	$l : x = \textbf{new } T$	ASSIGN	$l : x = y$
STORE	$l : x.f = y$	LOAD	$l : x = y.f$
THROW	$l : \textbf{throw } x$	CATCH	$l : \textbf{catch } y$
CALL	$l : x = a_0.f(a_1, ..., a_r)$	RETURN	$l : \textbf{return } ret$

Let $\mathbb{V}$, $\mathbb{H}$, $\mathbb{T}$, $\mathbb{M}$, $\mathbb{F}$, and $\mathbb{L}$ be the domains for representing sets of variables, heap abstractions, types, methods, field names, and statements (identified by their labels), respectively. Let $\mathbb{C}$ be the universe of contexts. The following auxiliary functions are used:

- $\texttt{PTS} : (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \to \wp(\mathbb{H} \times \mathbb{C})$
- $\texttt{MethodOf} : \mathbb{L} \to \mathbb{M}$
- $\texttt{Stmt} : \mathbb{M} \to \wp(\mathbb{L})$
- $\texttt{MethodCtx} : \mathbb{M} \to \wp(\mathbb{C})$
- $\texttt{VirtualCallDispatch} : \mathbb{M} \times \mathbb{H} \to \mathbb{M}$
- $\texttt{ExceptionDispatch} : \mathbb{L} \times \mathbb{H} \to \mathbb{L}$
- $\texttt{Cons} : \mathbb{H} \times \mathbb{C} \times \mathbb{L} \times \mathbb{C} \to \mathbb{C}$
- $\texttt{Sel} : (\mathbb{V} \cup \mathbb{H}) \times \mathbb{C} \mapsto \mathbb{C}$
- $\texttt{HeapAbs} : \mathbb{L} \times \mathbb{T} \mapsto \mathbb{H}$

where `PTS` records the points-to information found context-sensitively for a variable or an object's field, `MethodOf` gives the method containing a statement, `Stmt` returns the statements in a method, and `MethodCtx` maintains the contexts used for analyzing a method. As the pointer analysis is conducted together with both the call graph construction and exception analysis performed on the fly, we use `VirtualCallDispatch` to resolve a virtual call to a target method based on the dynamic type of the receiver object, and `ExceptionDispatch` to resolve a throwable statement to a catch statement by tracing the exception-catch links [7]. `Cons`, `Sel` and `HeapAbs` are three significant parameters described in Section 3.1. `Cons` describes how to construct a new context for a method, `Sel` selects some context elements from the context of a method to form a new context for a variable declared (or object allocated) in the method, and `HeapAbs` defines the heap abstraction for an object. We will discuss their instantiations for supporting different analysis algorithms in Section 4.

Figure 3 gives six rules used for formalizing our parameterized pointer analysis that supports both call graph construction and exception analysis on the fly. In [NEW], $o \in \mathbb{H}$ is an abstract heap object created by `HeapAbs`. Rules [ASSIGN], [LOAD] and [STORE] for handling assignments, loads and stores, respectively, are standard. In [THROW], a throwable object o pointed by variable x is dispatched to its corresponding catch trap along the exception-catch links [7]. In [CALL], a call to an instance method $x = a_0.f(a_1, ..., a_r)$ is analyzed. Here, $this^{m'}$, $p_i^{m'}$, and $ret^{m'}$ are the "this" variable, i-th parameter, and return variable of m', respectively, where m' is a target method resolved. In the conclusion of this rule, $ctx' \in \texttt{MethodCtx}(m')$ reveals how the contexts of a method are maintained. Initially, the contexts of all the entry methods are set to be empty, e.g., $\texttt{MethodCtx}(\text{"main"}) = \{[\,]\}$. To simulate Java's run-time

$$\frac{l : x = \text{new } T \quad o = \texttt{HeapAbs}(l, T) \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m)}{(o, \texttt{Sel}(o, ctx)) \in \texttt{PTS}(x, \texttt{Sel}(x, ctx))} \quad \text{[New]}$$

$$\frac{l : x = y \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m)}{\texttt{PTS}(y, \texttt{Sel}(y, ctx)) \subseteq \texttt{PTS}(x, \texttt{Sel}(x, ctx))} \quad \text{[Assign]}$$

$$\frac{l : x = y.f \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \quad (o, htx) \in \texttt{PTS}(y, \texttt{Sel}(y, ctx))}{\texttt{PTS}(o.f, htx) \subseteq \texttt{PTS}(x, \texttt{Sel}(x, ctx))} \quad \text{[Load]}$$

$$\frac{l : x.f = y \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \quad (o, htx) \in \texttt{PTS}(x, \texttt{Sel}(x, ctx))}{\texttt{PTS}(y, \texttt{Sel}(y, ctx)) \subseteq \texttt{PTS}(o.f, htx)} \quad \text{[Store]}$$

$$\frac{\begin{array}{c} l : \textbf{throw } x \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \\ (o, htx) \in \texttt{PTS}(x, \texttt{Sel}(x, ctx)) \quad l' : \textbf{catch } y = \texttt{ExceptionDispatch}(l, o) \end{array}}{(o, htx) \in \texttt{PTS}(y, \texttt{Sel}(y, ctx))} \quad \text{[Throw]}$$

$$\frac{\begin{array}{c} l : x = a_0.f(a_1, ..., a_r) \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \quad l' : \textbf{throw } t_l \in \texttt{Stmt}(m) \\ (o, htx) \in \texttt{PTS}(a_0, \texttt{Sel}(a_0, ctx)) \quad m' = \texttt{VirtualCallDispatch}(f, o) \quad ctx' = \texttt{Cons}(o, htx, l, ctx) \end{array}}{\begin{array}{c} ctx' \in \texttt{MethodCtx}(m') \quad (o, htx) \in \texttt{PTS}(this^{m'}, \texttt{Sel}(this^{m'}, ctx')) \\ \forall i \in [1, r] : \texttt{PTS}(a_i, \texttt{Sel}(a_i, ctx)) \subseteq \texttt{PTS}(p_i^{m'}, \texttt{Sel}(p_i^{m'}, ctx')) \\ \texttt{PTS}(eret^{m'}, \texttt{Sel}(eret^{m'}, ctx')) \subseteq \texttt{PTS}(t_l, \texttt{Sel}(t_l, ctx)) \quad \texttt{PTS}(ret^{m'}, \texttt{Sel}(ret^{m'}, ctx')) \subseteq \texttt{PTS}(x, \texttt{Sel}(x, ctx)) \end{array}} \quad \text{[Call]}$$

Figure 3 Rules for defining Qilin's parameterized pointer analysis.

semantic for re-throwing the exception objects (not handled in m') caused by its special catch statement "**catch** $eret'$", we associate a unique throw statement, l' : **throw** t_l, with the callsite l, where t_l is a special local variable in m for receiving these exception objects.

3.3 A High-Performance Incremental Worklist-based Solver

Doop [8] is declarative and uses a Datalog engine, e.g., Soufflé [20, 39] to compute the points-to facts for a pointer analysis. As Qilin is imperative, we have developed a new high-performance worklist-based constraint solver (depicted at ④ in Figure 2), which is currently single-threaded, for performing fine-grained context-sensitive pointer analyses (including variable-level context-sensitive pointer analyses in its full generality). When designing and implementing this constraint solver, we leverage an incremental worklist algorithm [23] suggested originally for resolving Andersen's context-insensitive pointer analysis [2]. However, we would like to stress that the basic algorithm used in our incremental worklist-based constraint solver is new, since variable-level context-sensitive pointer analyses require some points-to facts to be propagated in a way that does not exist traditionally before, as highlighted in Theorem 1. We will prove its correctness in Theorem 2 and illustrate its key part in supporting fine-grained context-sensitivity by using two examples (Tables 2 and 3).

Given a program P, we write `EntryOf(`P`)` to represent the set of its entry methods (including `main()`). To compute the points-to set $\texttt{PTS}(p, c)$ iteratively, where $p \in \mathbb{V} \cup \mathbb{H} \times \mathbb{F}$ and $c \in \mathbb{C}$, according to the rules given in Figure 3, we use four additional sets to represent four other kinds of context-sensitive facts that are also computed iteratively: (1) `PAG` (containing the currently discovered constraints expressed in terms of the assign, load and store edges in a PAG (Pointer Assignment Graph [23]), (2) `CALL` (containing the currently discovered call statements), (3) `THROW` (containing the currently discovered throw statements), and (4) `RM` (containing the (transitively) reachable methods found from `EntryOf(`P`)` so far).

For each of these five sets, denoted IS, we represent it as an *incremental set* [23] by dividing it into an "old" part ($\texttt{IS}_{\text{old}}$) and a "new" part ($\texttt{IS}_{\text{new}}$), so that $\texttt{IS} = \texttt{IS}_{\text{old}} \cup \texttt{IS}_{\text{new}}$, denoted also $\langle \texttt{IS}_{\text{old}}, \texttt{IS}_{\text{new}} \rangle$. At some point during the current iteration of an incremental worklist algorithm, `FlushNew(IS)` is called to flush $\texttt{IS}_{\text{new}}$ into $\texttt{IS}_{\text{old}}$, meaning that $\texttt{IS}_{\text{old}} \leftarrow \texttt{IS}_{\text{old}} \cup \texttt{IS}_{\text{new}}$ and $\texttt{IS}_{\text{new}} \leftarrow \varnothing$ are performed sequentially in that order. For notational convenience, we will write $S \overset{\cup}{\leftarrow} T$ as a shorthand for $S \leftarrow S \cup T$, where S and T are sets.

When computing the points-to facts iteratively for a pointer analysis, generalizing from method-level to variable-level context-sensitivity introduces one additional subtlety as summarized below, affecting the design of an incremental worklist-based constraint solver.

▶ **Theorem 1.** *Let method m (containing variable v) be analyzed under a new context c. Then $\texttt{PTS}(v, \texttt{Sel}(v, c))_{\text{old}} = \varnothing$ always holds under method-level context-sensitivity but $\texttt{PTS}(v, \texttt{Sel}(v, c))_{\text{old}} \neq \varnothing$ may hold under variable-level context-sensitivity.*

Proof. Let c' be a context under which m was analyzed earlier. Under method-level context-sensitivity, $\texttt{Sel}(v, c) = c$ and $\texttt{Sel}(v, c') = c'$. As $c \neq c'$, $\texttt{Sel}(v, c) \neq \texttt{Sel}(v, c')$. Hence, $\texttt{PTS}(v, \texttt{Sel}(v, c))_{\text{old}} = \varnothing$ (as $\langle v, \texttt{Sel}(v, c) \rangle$ is new and has never been analyzed before). Under variable-level context-sensitivity, $\texttt{Sel}(v, c) = \texttt{Sel}(v, c')$ may hold. Thus, $\texttt{PTS}(v, \texttt{Sel}(v, c))_{\text{old}} \neq \varnothing$ can hold if $\langle v, \texttt{Sel}(v, c) \rangle$ has been already analyzed earlier. ◀

Our incremental worklist-based constraint solver, given in Algorithm 1, takes a program P as input and applies the rules in Figure 3 to compute PTS as output, by performing both call graph construction and exception analysis on the fly. During the initialization (lines 1-5), RM is initialized with the entry methods in `EntryOf(`P`)`. `ProcessStmts` (lines 46-62) is called to initialize PTS, PAG, THROW and CALL with the points-to facts, three kinds of PAG edges, throw statements, and call statements found in these entry methods, respectively. Note that in line 61, l' : **throw** t_l is used for handling the exception objects thrown at callsite l, as discussed in Section 3.2. At this stage, the worklist W contains all the variables initialized to point to all the newly created objects (lines 48-50 ([New]) and lines 22-25).

The main loop (lines 6-20) discovers the points-to facts in the program iteratively. During each iteration, W contains a set of context-sensitive pointers p (variables or object fields) whose newly found points-to facts in $\texttt{PTS}(p)_{\text{new}}$ need to be propagated to their successors in PAG (i.e., $\texttt{PAG}_{\text{old}} \cup \texttt{PAG}_{\text{new}}$). Due to Theorem 1, however, for some context-sensitive pointer q that no longer appears in W, such that $o \in \texttt{PTS}(q)_{\text{old}}$ and f is a field of o, we may also need to propagate the points-to facts in $\texttt{PTS}(q)_{\text{old}}$ and/or $\texttt{PTS}(o.f)_{\text{old}}$ to their newly found successors in $\texttt{PAG}_{\text{new}}$ or from some newly found predecessors of $o.f$ in $\texttt{PAG}_{\text{new}}$ to $\texttt{PTS}(o.f)_{\text{new}}$.

During each iteration of the main loop (lines 6-20), we remove one such pointer *curr* from W and perform a four-step points-to fact propagation in the current iteration as follows:

- **Step 1: Resolving Direct Constraints (lines 8-11).** We resolve direct constraints by applying two rules in Figure 3: [Assign] (lines 8-9) and [Throw] (lines 10-11). In handling an outgoing assign edge at *curr*, we propagate the new points-to facts in $\texttt{PTS}(curr)_{\text{new}}$ to the successor of *curr* along this outgoing assign edge (lines 22-25). In handling a throw for *curr*, a thrown object is passed to the variable in its exception handler (lines 26-29).
- **Step 2: Resolving Indirect Constraints (lines 12-15).** We resolve indirect constraints by applying also two rules in Figure 3: [Load] (lines 12-13) and [Store] (lines 14-15). In handling load and store edges (lines 30-37), new assign edges are introduced in PAG (lines 33 and 37) to make their underlying assignment semantics explicit.
- **Step 3: Collecting New Constraints (lines 16-19).** We discover new reachable methods by first calling `HandleCall` to analyze the calls with *curr* as their receiver variable ([Call]) and then adding the new constraints for these new methods by calling

Algorithm 1 QILIN's incremental worklist-based constraint solver (implemented using incremental sets [23]) for supporting pointer analyses with fine-grained context-sensitivity.

```
    Input: P  // Input program
    Output: PTS  // Points-to Sets
2   ∀(v, c) ∈ 𝕍 × ℂ : PTS(v, Sel(v, c)) ← ⟨∅, ∅⟩
3   THROW ← CALL ← PAG ← ⟨∅, ∅⟩
4   W ← ∅
5   RM ← ⟨∅, {⟨m_e, [ ]⟩ | m_e ∈ EntryOf(P)}⟩
6   ProcessStmts ()
7   while W ≠ ∅ do
8       curr ← Poll(W)  // curr ∈ (𝕍 ∪ ℍ × 𝔽) × ℂ
        /* Step 1: Resolving Direct Constraints */
9       for curr --assign--> ⟨x, c⟩ ∈ PAG do
10          PropPTS(⟨x, c⟩, PTS(curr)_new)
11      for ⟨l : throw y, c⟩ ∈ THROW, s.t., curr = ⟨y, Sel(y, c)⟩ do
12          HandleThrow(l, curr, "new")
        /* Step 2: Resolving Indirect Constraints */
13      for curr --load[f]--> ⟨x, c⟩ ∈ PAG do
14          HandleLoad(curr --load[f]--> ⟨x, c⟩, "new")
15      for ⟨x, c⟩ --store[f]--> curr ∈ PAG do
16          HandleStore(⟨x, c⟩ --store[f]--> curr, "new")
        /* Step 3: Collecting New Constraints */
17      for ⟨l : x = a_0.f(⋯), c⟩ ∈ CALL, s.t., curr = ⟨a_0, Sel(a_0, c)⟩ do
18          HandleCall(l : x = a_0.f(⋯), c, "new")
19      ProcessStmts()
20      FlushNew(PTS(curr))
        /* Step 4: Activating New Constraints */
21      ActivateConstraints()
22  return PTS
23  Function PropPTS(⟨x, c⟩, s):
24      if ∃ ⟨o, htx⟩ ∈ s \ PTS(x, c) then
25          PTS(x, c)_new ←∪ s \ PTS(x, c)
26          W ←∪ {⟨x, c⟩}
27  Function HandleThrow(l, ⟨y, c⟩, i):
28      for ⟨o, htx⟩ ∈ PTS(y, c)_i do
29          l′ : catch x ← ExceptionDispatch(l, o)
30          PropPTS(⟨x, Sel(x, c)⟩, {⟨o, htx⟩})
31  Function HandleLoad(⟨y, c⟩ --load[f]--> ⟨x, c′⟩, i):
32      for ⟨o, htx⟩ ∈ PTS(y, c)_i do
33          if ⟨o.f, htx⟩ --assign--> ⟨x, c′⟩ ∉ PAG then
34              PAG_new ←∪ {⟨o.f, htx⟩ --assign--> ⟨x, c′⟩}
35  Function HandleStore(⟨y, c⟩ --store[f]--> ⟨x, c′⟩, i):
36      for ⟨o, htx⟩ ∈ PTS(x, c′)_i do
37          if ⟨y, c⟩ --assign--> ⟨o.f, htx⟩ ∉ PAG then
38              PAG_new ←∪ {⟨y, c⟩ --assign--> ⟨o.f, htx⟩}
39  Function HandleCall(l : x = a_0.f(a_1, ⋯, a_r), c, i):
40      for ⟨o, htx⟩ ∈ PTS(a_0, Sel(a_0, c))_i do
41          m′ ← VirtualCallDispatch(f, o), c′ ← Cons(o, htx, l, c)
42          PropPTS(⟨this^{m′}, Sel(this^{m′}, c′)⟩, {⟨o, htx⟩})
43          PAG_new ←∪ {⟨ret^{m′}, Sel(ret^{m′}, c′)⟩ --assign--> ⟨x, Sel(x, c)⟩}
44              ∪ {⟨eret^{m′}, Sel(eret^{m′}, c′)⟩ --assign--> ⟨t_l, Sel(t_l, c)⟩}
45              ∪ {⟨a_i, Sel(a_i, c)⟩ --assign--> ⟨p_i, Sel(p_i, c′)⟩ | i ∈ [1, r]}
46          RM_new ←∪ {⟨m′, c′⟩} \ RM
47  Function ProcessStmts():
48      for ⟨m, c⟩ ∈ RM_new do
49          for l : x = new T ∈ Stmt(m) do
50              o = HeapAbs(l, T)
51              PropPTS(⟨x, Sel(x, c)⟩, {⟨o, Sel(o, c)⟩})
52          for l : x = y ∈ Stmt(m) do
53              PAG_new ←∪ {⟨y, Sel(y, c)⟩ --assign--> ⟨x, Sel(x, c)⟩} \ PAG
54          for l : x = y.f ∈ Stmt(m) do
55              PAG_new ←∪ {⟨y, Sel(y, c)⟩ --load[f]--> ⟨x, Sel(x, c)⟩} \ PAG
56          for l : x.f = y ∈ Stmt(m) do
57              PAG_new ←∪ {⟨y, Sel(y, c)⟩ --store[f]--> ⟨x, Sel(x, c)⟩} \ PAG
58          for l : throw x ∈ Stmt(m) do
59              THROW_new ←∪ {⟨l : throw x, c⟩}
60          for l : x = a_0.f(a_1, ⋯, a_r) ∈ Stmt(m) do
61              CALL_new ←∪ {⟨l : x = a_0.f(a_1, ⋯, a_r), c⟩}
62              THROW_new ←∪ {⟨l′ : throw t_l, c⟩}
63      FlushNew(RM)
64  Function ActivateConstraints():
65      while CALL_new ≠ ∅ do
66          for ⟨l : x = a_0.f(a_1, ⋯, a_r), c⟩ ∈ CALL_new do
67              HandleCall(l : x = a_0.f(a_1, ⋯, a_r), c, "old")
68          FlushNew(CALL)
69          ProcessStmts()
70      for ⟨l : throw y, c⟩ ∈ THROW_new do
71          HandleThrow(l, ⟨y, Sel(y, c)⟩, "old")
72      FlushNew(THROW)
73      for ⟨y, c⟩ --load[f]--> ⟨x, c′⟩ ∈ PAG_new do
74          HandleLoad(⟨y, c⟩ --load[f]--> ⟨x, c′⟩, "old")
75      for ⟨y, c⟩ --store[f]--> ⟨x, c′⟩ ∈ PAG_new do
76          HandleStore(⟨y, c⟩ --store[f]--> ⟨x, c′⟩, "old")
77      for ⟨y, c⟩ --assign--> ⟨x, c′⟩ ∈ PAG_new do
78          PropPTS(⟨x, c′⟩, PTS(y, c)_old)
79      FlushNew(PAG)
80  Function FlushNew(IS):
81      ⟨IS_old, IS_new⟩ ← ⟨IS_old ∪ IS_new, ∅⟩
```

ProcessStmts. In lines 16-17, we process every call $\langle l : x = a_0.f(a_1, \cdots, a_r), c\rangle$ in CALL (i.e., both $\texttt{CALL}_{\text{old}}$ and $\texttt{CALL}_{\text{new}}$), where $curr = \langle a_0, \texttt{Sel}(a_0, c)\rangle$. In handling such a call (lines 38-45), new assign edges are introduced (lines 42-44) for modeling parameter passing, and in addition, new reachable methods are recorded in $\texttt{RM}_{\text{new}}$ (line 45). Note that $this^{m'}$ is handled (line 41) differently from the other parameters $p_1, ..., p_r$ (lines 42-44). A receiver object in a_0 flows only to the method dispatched on itself while the objects pointed to by the other arguments $a_1, ..., a_r$ flow to $p_1, ..., p_r$, respectively, for all the methods dispatched with a_0 as its receiver variable. For the new reachable methods just found, new constraints are added with PTS, PAG, THROW and CALL being updated (lines 46-62). Finally, in line 19, $\texttt{PTS}(curr)_{\text{new}}$ is flushed into $\texttt{PTS}(curr)_{\text{old}}$ as $curr$ has been processed.

- **Step 4: Activating New Constraints (line 20).** ActivateConstraints (lines 63-78) is called to initiate the points-to fact propagation across the new constraints in $\texttt{CALL}_{\text{new}}$, $\texttt{THROW}_{\text{new}}$ and $\texttt{PAG}_{\text{new}}$ found in Steps 2 – 3 by using the points-to facts in the "*old*" parts of the relevant pointers involved in these constraints. In lines 64-68, we process the calls in $\texttt{CALL}_{\text{new}}$ in turn by discovering more new reachable methods at every call $\langle l : x = a_0.f(a_1, \cdots, a_r), c\rangle$ in $\texttt{CALL}_{\text{new}}$ (by using $\texttt{PTS}(a_0, \texttt{Sel}(a_0, c))_{\text{old}}$ (line 66)) and adding the new constraints for the new reachable methods found (line 68) just after $\texttt{CALL}_{\text{new}}$ is flushed into $\texttt{CALL}_{\text{old}}$ (line 67). According to Theorem 1, lines 64-75 (shaded in blue) are needed to support variable-level context-sensitivity, as demonstrated by two examples below. Note that lines 76-77 are needed even in a context-insensitive analysis in order to support the on-the-fly call graph construction (among others) by activating the points-to propagation from the "old" part of an argument to its corresponding parameter in a newly found callee across its argument-to-parameter assign edge (line 44).

▶ **Theorem 2.** *Algorithm 1 computes the context-sensitive points-to information in a program exactly according to the rules given in Figure 3.*

Proof. We prove that Algorithm 1 computes the points-to facts according to the pointer analysis algorithm given in Figure 3 in the same manner. Thus, once a fixed point is reached, Algorithm 1 produces exactly the same points-to facts as the rules given in Figure 3.

- We first argue that during each iteration of Algorithm 1, one context-sensitive pointer $n \in (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}$ is removed from W and the objects in $\texttt{PTS}(n)_{\text{new}}$ are handled in exactly the same manner as in Figure 3: Step 1 handles [Assign] (lines 8-9), and [Throw] (lines 10-11). Step 2 handles [Load] (lines 12-13), and [Store] (lines 14-15). Step 3 handles [Call] (lines 16-17) and extends PAG with the newly reachable methods (line 18). Whenever an object allocation statement is visited, [New] is handled immediately (lines 48-50). Steps 2 and 3 serve only to add the newly discovered assign edges (constraints) to PAG without performing the actual points-to fact propagation. Step 4 activates these new constraints (lines 76-77). To support variable-level context-sensitivity according to Theorem 1, lines 64-75 (in blue) are added to activate also the newly reachable calls in $\texttt{CALL}_{\text{new}}$, throw statements in $\texttt{THROW}_{\text{new}}$, and loads/stores in $\texttt{PAG}_{\text{new}}$.
- We then argue that at the start of each iteration of Algorithm 1, $\forall n \in (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}$: $n \in \texttt{W} \iff \texttt{PTS}(n)_{\text{new}} \neq \varnothing$. "$\Longrightarrow$" is trivial by noting that n can only be added into W in line 25 and $\texttt{PTS}(n)_{\text{new}} \neq \varnothing$ due to line 23. We prove "$\Longleftarrow$" by induction. Initially, only the LHS of each allocation statement in the entry methods is added into W (line 5 and lines 48-50). Thus, "$\Longleftarrow$" holds. Given the induction hypothesis that "$\Longleftarrow$" holds at the start of the i-th iteration, we prove that "$\Longleftarrow$" still holds at the start of the $(i + 1)$-th iteration. During the i-th iteration, only $curr$ is removed from W at the start of the iteration and its points-to facts in $\texttt{PTS}(curr)_{\text{new}}$ have been flushed in line 19 after they have been handled. All the pointers that are added into W during this iteration must be added by PropPTS, ensuring that their new points-to facts are not empty.

By combining the two proof steps above, we conclude that for every n in $(\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}$, Algorithm 1 handles n in exactly the same manner as in Figure 3 for all the objects in $\texttt{PTS}(n)$. As a result, Algorithm 1 produces exactly the same points-to facts as Figure 3. ◀

Below we use two small example programs to illustrate how our worklist-based constraint solver (Algorithm 1) works in supporting pointer analyses with variable-level context-sensitivity. In addition, we will also highlight the two subtleties involved (one in each example) in designing this new constraint solver for computing the points-to facts iteratively.

In the first example, we explain how our constraint solver works in computing the points-to facts for the program in Figure 1, by applying the 1-object-sensitive pointer analysis with variable-level context-sensitivity discussed in Section 2.3 under which only the parameter `x` of `foo()` is analyzed context-sensitively. We would like to stress the significance of Theorem 1 by highlighting the necessity of lines 64-75 (shaded in blue) in supporting variable-level context-sensitivity. As `foo()` is called under two different receiver objects in lines 13 and 16, `x` will be qualified by either `[B1]` or `[B2]`. Every other variable/object p, which is analyzed context-insensitively, is identified by $\langle p, [\,]\rangle$, which will be abbreviated to p for brevity. Therefore, whenever we write $\texttt{PTS}(p)$ without providing a context, we mean $\texttt{PTS}(p, [\,])$.

Table 2 traces one particular execution of Algorithm 1, showing how `PTS`, `RM`, `CALL`, `PAG`, and `W` are updated incrementally in a total of 17 iterations (with its initialization assumed to start at 0). For this simple program. `THROW` is not relevant. To save space, we have segmented these 17 iterations into six groups. For each group, we start with `W` being given in the preceding group at the beginning of its first iteration and produce the results obtained for `PTS`, `RM`, `CALL`, `PAG`, and `W` at the end of its last iteration. For `PTS`, we list explicitly both the "old" and "new" parts for all its variables and fields. For `RM`, `CALL`, and `PAG`, we list only their "new" parts as their "old" parts can be read-off easily from the earlier iterations given.

- **Iteration 0**. Initially, we perform the initialization (lines 1-5) by taking `main()` as the only entry of the program. Then we compute the points-to information iteratively during all the iterations of the `while` loop in line 6 (i.e., Iterations 1–16).
- **Iterations 1-3**. We start with the worklist `W` given at Iteration 0 and then obtain the updated results as shown after `o1`, `o2` and `a1` have been processed during these three iterations. At this point, $\texttt{CALL}_{\text{new}}$ has already been flushed into $\texttt{CALL}_{\text{old}}$ so that $\texttt{CALL} = \langle \texttt{CALL}_{\text{old}}, \texttt{CALL}_{\text{new}}\rangle = \langle\{\langle \texttt{line 13}, [\,]\rangle, \langle \texttt{line 16}, [\,]\rangle\}, \varnothing\rangle$.
- **Iteration 4**. We start with $curr$ = "b1", where $\texttt{PTS(b1)}_{\text{new}} = \{\texttt{B1}\}$, indicating that we are just about to analyze `foo()` under context `[B1]` (invoked in line 13 in the program). In Step 3, a total of five edges are added to $\texttt{PAG}_{\text{new}}$. The three new assign edges are introduced for modeling parameter passing for this particular call (two for the two parameters `x` and `a`, and one for the return variable `x`). In addition, the store edge and the load edge are introduced for representing the two statements in `foo()`. At the end of Step 3, $\texttt{PTS(b1)}_{\text{new}}$ is flushed into $\texttt{PTS(b1)}_{\text{old}}$. In Step 4, the propagation into the two parameters, $\langle \texttt{x}, [\texttt{B1}]\rangle$ and `a`, from their corresponding actual arguments is made (lines 76-77).
- **Iterations 5-10**. We propagate the newly found points-to information by processing `a`, $\langle \texttt{x}, [\texttt{B1}]\rangle$, and $\texttt{this}^{\texttt{foo}}$ in `W` and the others added later to `W` iteratively.
- **Iteration 11**. We start with $curr$ = "b2", where $\texttt{PTS(b2)}_{\text{new}} = \{\texttt{B2}\}$, so that we can analyze `foo()` under context `[B2]` (invoked in line 16 in the program). We proceed exactly as when `foo()` is analyzed under context `[B1]` at iteration 4 except that we no longer need to introduce $\texttt{a1} \xrightarrow{\text{assign}} \texttt{a}$ and $\texttt{a} \xrightarrow{\text{load[f]}} \texttt{t}$ in Step 3, since both edges already exist in `PAG`. Before Step 4 starts, $\texttt{PTS(b2)}_{\text{new}}$ has already been flushed into $\texttt{PTS(b2)}_{\text{old}}$. In Step 4,

Table 2 Tracing a particular execution of Algorithm 1 in applying the fine-grained 1-object-sensitive pointer analysis discussed in Section 2.3 to the program given in Figure 1.

Iters	$\mathbb{V}$	PTS_{old}	PTS_{new}	RM_{new}	CALL_{new}	PAG_{new}	W	Points-to Fact Propagation
0	o1 o2 b1 b2 a1	∅ ∅ ∅ ∅ ∅	{O1} {O2} {B1} {B2} {A1}	{⟨main, [ ]⟩}	{⟨line 13, [ ]⟩, ⟨line 16, [ ]⟩}	∅	{o1, o2, b1, b2, a1}	A1 → a1 O1 → o1 O2 → o2 B1 → b1 B2 → b2
1-3	o1 o2 b1 b2 a1	{O1} {O2} ∅ ∅ {A1}	∅ ∅ {B1} {B2} ∅	∅	∅	∅	{b1, b2}	SAME AS ABOVE
4	o1 o2 b1 b2 a1 this^{foo} ⟨x, B1⟩ a	{O1} {O2} {B1} ∅ {A1} ∅ ∅ ∅	∅ ∅ ∅ {B2} ∅ {B1} {O1} {A1}	{⟨foo, [B1]⟩}	{⟨line 7, [B1]⟩}	{o1 $\xrightarrow{\text{assign}}$ ⟨x, [B1]⟩, a1 $\xrightarrow{\text{assign}}$ a, ⟨x, [B1]⟩ $\xrightarrow{\text{assign}}$ v1, ⟨x, [B1]⟩ $\xrightarrow{\text{store[f]}}$ a, a $\xrightarrow{\text{load[f]}}$ t}	{b2, a, ⟨x, [B1]⟩, this^{foo}}	O1 → o1 → ⟨x, [B1]⟩ → v1 A1 → a1 → a O2 → o2 B1 → b1 → this^{foo} B2 → b2
5-10	o1 o2 b1 b2 a1 v1 this^{foo} ⟨x, [B1]⟩ a A1.f t	{O1} {O2} {B1} ∅ {A1} {O1} {B1} {O1} {A1} {O1} {O1}	∅ ∅ ∅ {B2} ∅ ∅ ∅ ∅ ∅ ∅ ∅	∅	∅	{⟨x, [B1]⟩ $\xrightarrow{\text{assign}}$ A1.f, A1.f $\xrightarrow{\text{assign}}$ t}	{b2}	O1 → o1 → ⟨x, [B1]⟩ → v1 ⟨x, [B1]⟩ → A1.f → t A1 → a1 → a O2 → o2 B1 → b1 → this^{foo} B2 → b2
11	o1 o2 b1 b2 a1 v1 this^{foo} ⟨x, [B1]⟩ ⟨x, [B2]⟩ a A1.f t	{O1} {O2} {B1} {B2} {A1} {O1} {B1} {O1} ∅ {A1} {O1} {O1}	∅ ∅ ∅ ∅ ∅ ∅ {B2} ∅ {O2} ∅ {} ∅	{⟨foo, [B2]⟩}	{⟨line 7, [B2]⟩}	{o2 $\xrightarrow{\text{assign}}$ ⟨x, [B2]⟩, ⟨x, [B2]⟩ $\xrightarrow{\text{assign}}$ v2, ⟨x, [B2]⟩ $\xrightarrow{\text{store[f]}}$ a ⟨x, [B2]⟩ $\xrightarrow{\text{assign}}$ A1.f}	{this^{foo}, ⟨x, [B2]⟩}	O1 → o1 → ⟨x, [B1]⟩ → v1 ⟨x, [B1]⟩ → A1.f → t A1 → a1 → a O2 → o2 → ⟨x, [B2]⟩ → v2 ⟨x, [B2]⟩ → A1.f B1 → b1 → this^{foo} B2 → b2 → this^{foo}
12-16	o1 o2 b1 b2 a1 v1 v2 this^{foo} ⟨x, [B1]⟩ ⟨x, [B2]⟩ a A1.f t	{O1} {O2} {B1} {B2} {A1} {O1} {O2} {B1, B2} {O1} {O2} {A1} {O1, O2} {O1, O2}	∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅	∅	∅	∅	∅	SAME AS ABOVE

the points-to information of ⟨x, [B2]⟩ is updated (lines 76-77). In addition, we discover $\langle x, [B2]\rangle \xrightarrow{\text{assign}} \langle A1.f, [\,]\rangle$ from $\langle x, [B2]\rangle \xrightarrow{\text{store[f]}} \langle a, [\,]\rangle$ in lines 74-75 (Theorem 1), since $\texttt{PTS}(a, [\,])_{old} = \langle A1, [\,]\rangle$. Otherwise, O2 in PTS(t) will be missed unsoundly.

- **Iterations 12-16**. After these iterations have been done, a fixpoint will be reached.

In Algorithm 1, its `while` loop in line 64 is needed for supporting a number of different flavors of context-sensitivity. Next, we use a second example to explain its necessity for supporting k-callsite-based context-sensitive pointer analysis (i.e., kCFA [41]) with variable-level context-sensitivity. This loop is needed to avoid a rare case under which W = ∅ but $\texttt{CALL}_{new} \neq \emptyset$, implying that the new reachable methods in $\texttt{CALL}_{new}$ must still need to be analyzed to look for the new points-to facts despite the fact that the worklist W is empty.

In this second example given in Figure 4, three classes, A, B, and C, are defined, where `create()` and `wcreate()` in B and `wcreate2()` in C are used to establish a chain of method calls in the program, where `wcreate2()` is a wrapper method for `wcreate1()`, which is a wrapper method for `create()`. In `main()`, O1 is first allocated, then stored into `a1.f`, and finally, retrieved from `a1.f` and assigned to `v1`. Similarly, O2 is first allocated, then stored into `a2.f`, and finally, retrieved from `a2.f` and assigned to `v2`.

```
1  void main() {
2    C c1 = new C(); // C1
3    C c2 = c1;
4    A a1 = c1.wcreate2(); // c1
5    A a2 = c2.wcreate2(); // c2
6    Object o1 = new Object(); // O1
7    Object o2 = new Object(); // O2
8    a1.f = o1;
9    a2.f = o2;
10   Object v1 = a1.f;
11   Object v2 = a2.f;
12 }
13
14 class A { Object f; }
15 class B {
16   A create() {
17     A r1 = new A(); // A1
18     return r1; }
19   A wcreate() {
20     A r2 = this.create(); // c4
21     return r2;
22 }}
23 class C {
24   A wcreate2() {
25     B b1 = new B(); // B1
26     A r3 = b1.wcreate(); // c3
27     return r3;
28 }}
```

Figure 4 An example for illustrating the necessity of the `while` loop in line 64 of Algorithm 1 for supporting callsite-based context-sensitive pointer analyses with variable-level context-sensitivity.

Let us analyze this program by using 4CFA with variable-level context-sensitivity, under which `r1`, `r2`, `r3` and A1 are context-sensitive but all the remaining variables and objects are context-insensitive. In particular, as is usually done in practice [22, 14, 25], the length of a context for `r1`, `r2` and `r3` is limited by 4 and the length of a context for A1 is limited by 3.

If we apply our constraint solver (Algorithm 1) to solve this particular 4CFA-style pointer analysis for this program, its points-to facts will be computed soundly as desired. In particular, `v1` will be found to point to O1 and `v2` will be found to point to O2. However, if we apply our constraint solver with its line 64 being deleted, then the resulting modified constraint solver will be unsound. For this particular program, `a2` will be found point to no object at all, and consequently, that `v2` will also be regarded as pointing to no object at all.

Table 3 traces a particular execution of this modified constraint solver (in 15 iterations):

- **Iterations 0-2**. During the initialization, i.e., at Iteration 0 (lines 1-5), `main()` is the only entry method for the program. By processing its statements, we end up adding two new calls (for its lines 4-5) to $\texttt{CALL}_{new}$, five new edges (for its lines 3 and 8-11) to $\texttt{PAG}_{new}$ and {c1, o1, o2} (for its lines 2 and 6-7) to W. At iterations 1-2, o1 and o2 are handled by just flushing $\texttt{PTS}(o1)_{new}$ and $\texttt{PTS}(o2)_{new}$ into $\texttt{PTS}(o1)_{old}$ and $\texttt{PTS}(o2)_{old}$, respectively.

Table 3 Tracing a particular execution of Algorithm 1 *with its* `while` *loop in line 64* being deleted in applying 4CFA with variable-level context-sensitivity to the program given in Figure 4, under which all the variables and objects except `r1`, `r2`, `r3`, and `A1` are context-insensitive.

Iters	$\mathbb{V}$	PTS_{old}	PTS_{new}	RM_{new}	CALL_{new}	PAG_{new}	W
0-2	c1	∅	{C1}	{⟨main, [ ]⟩}	{⟨line 4, [ ]⟩, ⟨line 5, [ ]⟩}	{c1 $\xrightarrow{\text{assign}}$ c2 o1 $\xrightarrow{\text{store[f]}}$ a1 o2 $\xrightarrow{\text{store[f]}}$ a2 a1 $\xrightarrow{\text{load[f]}}$ v1 a2 $\xrightarrow{\text{load[f]}}$ v2}	{c1}
	o1	{O1}	∅				
	o2	{O2}	∅				
3-13	c1	{C1}	∅	{⟨wcreate2, [c1]⟩ ⟨wcreate, [c3, c1]⟩ ⟨create, [c4, c3, c1]⟩}	{⟨line 26, [c1]⟩ ⟨line 20, [c3, c1]⟩}	{⟨r3, [c1]⟩ $\xrightarrow{\text{assign}}$ a1 ⟨r2, [c3, c1]⟩ $\xrightarrow{\text{assign}}$ ⟨r3, [c1]⟩ ⟨r1, [c4, c3, c1]⟩ $\xrightarrow{\text{assign}}$ ⟨r2, [c3, c1]⟩ o1 $\xrightarrow{\text{assign}}$ ⟨A1.f, [c4, c3, c1]⟩ ⟨A1.f, [c4, c3, c1]⟩ $\xrightarrow{\text{assign}}$ v1}	{c2}
	o1	{O1}	∅				
	o2	{O2}	∅				
	c2	∅	{C1}				
	$\text{this}^{\text{wcreate2}}$	{C1}	∅				
	b1	{B1}	∅				
	$\text{this}^{\text{wcreate}}$	{B1}	∅				
	$\text{this}^{\text{create}}$	{B1}	∅				
	⟨r1, [c4, c3, c1]⟩	{⟨A1, [c4, c3, c1]⟩}	∅				
	⟨r2, [c3, c1]⟩	{⟨A1, [c4, c3, c1]⟩}	∅				
	⟨r3, [c1]⟩	{⟨A1, [c4, c3, c1]⟩}	∅				
	a1	{⟨A1, [c4, c3, c1]⟩}	∅				
	⟨A1.f, [c4, c3, c1]⟩}	{O1}	∅				
	v1	{O1}	∅				
14	c1	{C1}	∅	{⟨wcreate2, [c2]⟩ ⟨wcreate, [c3, c2]⟩}	{⟨line 26, [c2]⟩ ⟨line 20, [c3, c2]⟩}	{⟨r3, [c2]⟩ $\xrightarrow{\text{assign}}$ a2 ⟨r2, [c3, c2]⟩ $\xrightarrow{\text{assign}}$ ⟨r3, [c2]⟩}	
	o1	{O1}	∅				
	o2	{O2}	∅				
	c2	{C1}	∅				
	$\text{this}^{\text{wcreate2}}$	{C1}	∅				
	b1	{B1}	∅				
	$\text{this}^{\text{wcreate}}$	{B1}	∅				
	$\text{this}^{\text{create}}$	{B1}	∅				
	⟨r1, [c4, c3, c1]⟩	{⟨A1, [c4, c3, c1]⟩}	∅				
	⟨r2, [c3, c1]⟩	{⟨A1, [c4, c3, c1]⟩}	∅				
	⟨r3, [c1]⟩	{⟨A1, [c4, c3, c1]⟩}	∅				
	a1	{⟨A1, [c4, c3, c1]⟩}	∅				
	⟨A1.f, [c4, c3, c1]⟩}	{O1}	∅				
	v1	{O1}	∅				

- **Iterations 3-13**. We propagate the newly found points-to information by processing `c1` and others added later to `W` during these few iterations iteratively while keeping `c2` always in `W`. This particular execution order is possible, since the items (i.e., unprocessed pointers) in `W` are processed non-deterministically. At the end of Iteration 13, the points-to information for all the variables except `a2` and `v2` has been obtained.
- **Iteration 14**. We start with *curr* = "c2", where $\text{PTS}(\text{c2})_{\text{new}} = \{\text{C1}\}$, indicating that we are just about to analyze `wcreate2()` under context [c2] (invoked in line 5 in the program). In Step 3, ⟨r3, [c2]⟩ $\xrightarrow{\text{assign}}$ a2 is added to PAG_{new} and a new call in line 26 in the program invoked under context [c2] is discovered and added into CALL_{new}. We no longer add $\text{this}^{\text{wcreate2}}$ and `b1` to `W` as they do not point to any new points-to fact discovered. In Step 4, we handle the new call (line 26 in the program under [c2]) in CALL_{new} with $\text{PTS}(b1)_{\text{old}} = \{\text{B1}\}$ (lines 65-66) and find a new reachable method `wcreate()` under context [c3, c2]. Finally, we call `ProcessStmts` (line 68) to process the statements in this newly reachable method, establish ⟨r2, [c3, c2]⟩ $\xrightarrow{\text{assign}}$ ⟨r3, [c2]⟩, and discover one more call (line 20 in the program) under context [c3, c2] (highlighted in red). In the modified constraint solver, the `while` loop in line 64 (Algorithm 1) has been deleted. As `W` = ∅, this new call will not be processed during the next iteration. As a result, `a2` and `v2` will be concluded not to point to any object unsoundly. However, our constraint solver, with this `while` loop being used in line 64, works soundly.

3.4 Handling Complex Language Features

We have closely modeled the handling of complex Java features and semantic complexities after the logic in DOOP [8] (as depicted at ⑤ in Figure 2), so that QILIN achieves exactly the same precision for a program as DOOP except for a few tool-specific variables introduced.

We have modeled the system/main thread groups and main thread to identify a variety of the entry methods of a Java program (line 4 of Algorithm 1), Java's reference objects (e.g., WeakReference and SoftReference) and reference queues, and class initialization. For example, JVM will register reference objects to reference queues by calling `Finalizer.register()` so that `finalize()` methods can be invoked. In addition, JVM will initialize classes/interfaces by calling their static initializers, `<clinit>()`. To handle such implicit calls by JVM, we dynamically inject static calls into the body of a `FakeMain()` method (regarded as an entry of the program) to simulate their behavior during the pointer analysis for a given Java program.

To model a native method, we have designed a *handler* to simulate its semantics by generating a method body in Jimple [54], the IR (Intermediate Representation) for SOOT. Currently, QILIN handles the same set of native methods (e.g., `thread:start()`, `DoPrivileged()` and `clone()`) supported by DOOP in exactly the same way.

As for Java reflection, DOOP can handle it either statically or dynamically (by relying on the reflective targets found by TAMIFLEX [6]). We have taken the latter approach since it has been used exclusively in the pointer analysis community in the past few years [48, 25, 30, 14]. QILIN's reflection handler supports exactly the same set of the most commonly used Java reflection APIs (e.g., `ClassForName()` and `ClassNewInstance()`) as in DOOP.

QILIN, as in DOOP, handles cast and assignment compatibility by using the declared type of a variable to filter out type-incompatible pointed-to objects during the pointer analysis. As is standard, arrays are considered monolithic (without distinguishing their elements). In particular, we filter out type-incompatible objects stored in an array by using the declared type of its elements instead of `java.lang.Object`.

In QILIN, we handle static fields and static methods in the standard manner. As global variables, static fields are analyzed context-insensitively. A static method `m()` is modeled as a special instance method by just interpreting a call to `m()` as `this.m()` and proceeding as if it were an instance method defined in `java.lang.Object`. As a result, static methods can be analyzed uniformly under all flavors of context-sensitivity except for hybrid context-sensitivity [22], under which static and virtual calls are distinguished.

4 Using the Qilin Framework

We first describe a few significant instantiations of `Cons` (Section 4.1), `Sel` (Section 4.2), and `HeapAbs` (Section 4.3), respectively. We then combine these instantiations to obtain the pointer analyses provided in QILIN's toolbox depicted at ① in Figure 2 (Section 4.4).

Given a context $c = [e_1, \cdots, e_n]$ and a context element e_0, we write $e_0 +\!\!+ c$ to denote $[e_0, e_1, \cdots, e_n]$, and $\lceil c \rceil_k$ for $[e_1, \cdots, e_k]$ (i.e., c restricted to its prefix of length k).

4.1 Context Constructors

We instantiate $\texttt{Cons}(o, htx, l, ctx)$ used in [CALL] (Figure 3) to define five common types of contexts for a method ("Insens", "Callsite", "Object", "Type", and "Hybrid" listed at ② in Figure 2). Note that in our framework, k-limiting will be applied by `Sel`.

- Insens. For a context-insensitive pointer analysis [2, 23], all methods are analyzed under the same fixed empty context (without distinguishing their calling contexts):

$$\texttt{Cons}_{\text{INSENS}}(o, htx, l, ctx) = [\,] \tag{1}$$

- Callsite. A callsite-sensitive pointer analysis [40], known also as *control-flow analysis* (CFA) [41], uses a callsite l as a context element. Therefore, the context constructor is:

$$\texttt{Cons}_{\text{CFA}}(o, htx, l, ctx) = l +\!\!+\, ctx \tag{2}$$

- Object. An object-sensitive pointer analysis [32, 33] uses a receiver object o as a context element. Thus, the context constructor simply becomes:

$$\texttt{Cons}_{\text{OBJ}}(o, htx, l, ctx) = o +\!\!+\, htx \tag{3}$$

- Type. A type-sensitive pointer analysis [42], which is a more scalable but less precise alternative of an object-sensitive pointer analysis, resorts to the class type containing the method where a receiver object o is allocated, denoted as $\texttt{TypeContg}(o)$. Thus, we have:

$$\texttt{Cons}_{\text{TYPE}}(o, htx, l, ctx) = \texttt{TypeContg}(o) +\!\!+\, htx \tag{4}$$

- Hybrid. A hybrid pointer analysis [22] distinguishes static and dynamic call sites:

$$\texttt{Cons}_{\text{HYB}}(o, htx, l, ctx) = \begin{cases} o +\!\!+\, htx & l \notin \texttt{SC} \\ \texttt{car}(ctx) +\!\!+\, l +\!\!+\, \texttt{cdr}(ctx) & l \in \texttt{SC} \end{cases} \tag{5}$$

  where `SC` is the set of all static call sites in the program. Here, `car` and `cdr` are standard, with `car` pulling the first element of a list and `cdr` returning the list without the `car`. For a non-static callsite, the new context is constructed identically as in Equation (3). For a static callsite, the new context also includes the invocation site, which has been shown to be effective in improving the precision of pointer analysis [22].

4.2 Context Selectors

It is simple to instantiate a context selector `Sel` to support both method-level and variable-level context-sensitivity, including "Uniform", "Heuristic", "Selective", and "Partial" listed at ② in Figure 2. Given a calling context for a method, a context selector `Sel` picks some of its context elements to define the contexts for the variables/objects in the method by applying k-limiting [41], where k can vary across the variables/objects in the same method.

- Uniform. To support traditional method-level context-sensitive pointer analyses that rely on k-limited context abstractions [40, 32, 42, 22], a "uniform" context selector is used:

$$\texttt{Sel}_{\mathcal{U}}(ctx, e) = \begin{cases} \lceil ctx \rceil_k & e \in \mathbb{V} \\ \lceil ctx \rceil_{hk} & e \in \mathbb{H} \end{cases} \tag{6}$$

  where the local variables (objects allocated) in a method adopt its calling context ctx uniformly under k-limiting (hk-limiting). In practice, $hk = k - 1$ is often used.
- Heuristic. For efficiency reasons at little loss of precision, the objects of certain types from an empirically determined set, $\mathcal{T}$, are usually analyzed context-insensitively as in Doop [8] and Wala [16], where the following definition of $\mathcal{T}$ is the most popularly used:

$$\mathcal{T} = \{\texttt{StringBuffer}, \texttt{StringBuilder}, \texttt{Throwable}\} \tag{7}$$

Let $\mathtt{SubTypeOf}(\mathcal{T})$ be the set including the types in $\mathcal{T}$ and their subtypes. Let $\mathtt{TypeOf}(o)$ be the dynamic type of an object o. A "*heuristic*" context selector is given by:

$$\mathtt{Sel}_{\mathcal{H}}(ctx, e) = \begin{cases} [\,] & e \in \mathbb{H} \land \mathtt{TypeOf}(e) \in \mathtt{SubTypeOf}(\mathcal{T}) \\ \mathtt{Sel}_{\mathcal{U}}(ctx, e) & \text{otherwise} \end{cases} \tag{8}$$

- Selective. Under "*selective*" method-level context-sensitivity [43, 25], only a subset of (precision-critical) methods, CSML, in the program is selected to be analyzed context-sensitively:

$$\mathtt{Sel}_{\mathcal{S}}(ctx, e) = \begin{cases} [\,] & \mathtt{MethodOf}(e) \notin \textsc{Csml} \\ \mathtt{Sel}_{\mathcal{U}}(ctx, e) & \mathtt{MethodOf}(e) \in \textsc{Csml} \end{cases} \tag{9}$$

- Partial. Under "*partial*" method-level context-sensitivity [30, 14], only a subset of (precision-critical) variables/objects in the program, CPML, is selected to be analyzed context-sensitively:

$$\mathtt{Sel}_{\mathcal{P}}(ctx, e) = \begin{cases} [\,] & e \notin \textsc{Cpml} \\ \mathtt{Sel}_{\mathcal{U}}(ctx, e) & e \in \textsc{Cpml} \end{cases} \tag{10}$$

The power of QILIN goes beyond existing fine-grained context-sensitive pointer analyses [43, 25, 30, 14]. In our pointer analysis framework, different variables/objects can be analyzed completely independently under different context abstractions, thus providing support for fine-grained context selectivity in its full generality.

4.3 Heap Abstractors

We can instantiate $\mathtt{HeapAbs}(l, T)$ in [NEW] (Figure 3) to define a range of heap abstractions used, including "Allocation-Site", "Heuristic", and "Type-Consistency" listed at ② in Figure 2.

- Allocation-Site. This represents the most widely used heap abstraction:

$$\mathtt{HeapAbs}_{\mathcal{A}}(l, T) = O_l \tag{11}$$

  where O_l is an abstract object created at the allocation site identified by its label l.
- Heuristic. In practice, for efficiency reasons at little loss of precision, the objects of a particular type may be distinguished per dynamic type (instead of per object). As a result, we obtain the following "*heuristic*" heap abstractor (with $\mathcal{T}$ given in Eq. (7)):

$$\mathtt{HeapAbs}_{\mathcal{H}}(l, T) = \begin{cases} O_T & T \in \mathtt{SubTypeOf}(\mathcal{T}) \\ O_l & \text{otherwise} \end{cases} \tag{12}$$

- Type-Consistency. For the "*type-consistent*" heap abstraction proposed in [48], we have:

$$\mathtt{HeapAbs}_{\mathcal{T}}(l, T) = \mathtt{rep}(S(O_l)) \tag{13}$$

  where $S(O_l)$ is the equivalence class containing the objects that are type-consistent as O_l and $\mathsf{rep}(S(O_l))$ is its representative. In other words, all the allocation sites are divided into equivalence classes so that those in the same equivalence class are not distinguished.

4.4 Qilin's Toolbox

Qilin, as shown at ① in Figure 2, includes already a rich set of pointer analyses for supporting (1) *insens* (Andersen's context-insensitive analysis) [23], (2) all common flavors of method-level context-sensitivity: kCFA (k-callsite-sensitivity) [40], kOBJ (k-object-sensitivity) [32, 33], kTYPE (k-type-sensitivity) [42], S-kOBJ (hybrid k-object-sensitive analysis) [22], and (3) many flavors of fine-grained context-sensitivity, enabled by different pre-analyses (for defining `Sel` and `HeapAbs`): Bean [49], Mahjong [48], Zipper [25], Eagle [30], Turner [14], Conch [15], Data-Driven [19], and Context-Tunneling [17].

Table 4 lists a subset of these analyses (evaluated below) and their instantiations. Given two context selectors s_1 and s_2, we define $\texttt{Min}(s_1, s_2) = \lambda\,(ctx, e).\ \textbf{if}\ |s_1(ctx, e)| \leqslant |s_2(ctx, e)|\ \textbf{then}\ s_1(ctx, e)\ \textbf{else}\ s_2(ctx, e)$. Each analysis is specified by a triple $[\texttt{Cons}, \texttt{Sel}, \texttt{HeapAbs}]$. Z-$k$OBJ, E-$k$OBJ, and T-$k$OBJ are the versions of kOBJ performed with fine-grained context-sensitivity prescribed by Zipper [25], Eagle [30], and Turner [14], respectively.

Table 4 A subset of pointer analyses instantiated in Qilin.

Pointer Analysis	Instantiation (Parameterization)
insens [23]	[Eq. (1), Eq. (8), Eq. (12)]
kCFA [40]	[Eq. (2), Eq. (8), Eq. (12)]
kOBJ [32, 33]	[Eq. (3), Eq. (8), Eq. (12)]
S-kOBJ [22]	[Eq. (5), Eq. (8), Eq. (12)]
Z-kOBJ [25]	[Eq. (3), `Min`(Eq. (9), Eq. (8)), Eq. (12)]
E-kOBJ [30]	[Eq. (3), `Min`(Eq. (10), Eq. (8)), Eq. (12)]
T-kOBJ [14]	[Eq. (3), `Min`(Eq. (10), Eq. (8)), Eq. (12)]

5 Evaluation

We have implemented Qilin as a standalone tool in Java in 20.3 KLOC (including 4.7 KLOC at its core) that operates on the Jimple IR [54] of Soot (version `4.2.1`) [53]. Qilin (including a micro-benchmark suite consisting of ≈ 100 unit test cases) has been open-sourced and maintained at `https://github.com/QiLinPTA/QiLin`.

Our evaluation aims to show that Qilin has met all its four design goals by answering the following four questions positively:

- **RQ1.** Is Qilin precise in terms of the precision achieved against the state-of-the-art?
- **RQ2.** Is Qilin efficient in terms of the analysis time taken against the state-of-the-art?
- **RQ3.** Is Qilin modular in allowing its common codebase to be shared?
- **RQ4.** Is Qilin effective in supporting fine-grained context-sensitive pointer analyses?

We report and analyze our results by focusing on the seven representative analyses listed in Table 4, *insens*, 1CFA, 2OBJ and S-2OBJ (with method-level context-sensitivity) and E-2OBJ, T-2OBJ and Z-2OBJ (with fine-grained context-sensitivity). We address **RQ1** and **RQ2** by comparing Qilin with Doop [8] (using a recent stable version `4.24.0` with Soufflé Datalog engine 1.5.1 [20]) in supporting *insens*, 1CFA, 2OBJ and S-2OBJ. Note that Doop has been tested with Soufflé 1.5.1 and Soufflé 2.0.2, but Doop is slower under Soufflé 2.0.2 than under Soufflé 1.5.1 in our evaluation. During this process, we have fixed a number of bugs in Doop that may have caused its unsoundness as also reported earlier [37]. We address **RQ3** and **RQ4** by considering how Qilin supports E-2OBJ, T-2OBJ and Z-2OBJ (among others).

We have used a large Java library (`JDK1.6.0_45`) and 12 popular Java programs (including 9 benchmarks from DaCapo 2006 [5] and 3 Java applications, `checkstyle`, `JPC` and `findbugs`), which are frequently used for evaluating pointer analysis algorithms in the literature [43,

22, 48, 25, 14, 28, 15]. We have excluded `jython` and `hsqldb` since their context sensitive analyses do not scale due to overly conservative handling of Java reflection [50]. By using DaCapo 2006 as in these earlier papers, we are able to evaluate these earlier algorithms in QILIN with reference to the results reported earlier. We have carried out all the experiments on an eight-core Intel(R) Xeon(R) CPU E5-2637 3.5GHz machine with 512GB of RAM.

5.1 RQ1: Precision

As shown in Table 5, QILIN delivers exactly the same precision as DOOP for *insens*, 1CFA, 2OBJ and S-2OBJ, since both tools (1) use the same logic points-to definitions (Section 3.3), and (2) cover the same complex Java features identically (Section 3.4). The precision of a pointer analysis is measured in terms of four common metrics [42, 48, 30, 25]: (1) *#call-edges*: the number of call graph edges discovered, (2) *#fail-cast*: the number of type casts that may fail, (3) *#poly-calls*: the number of polymorphic calls discovered, and (4) *#avg-pts*: the average number of objects pointed by a variable, i.e., the average points-to set size by considering only the variables in the Java methods (i.e., excluding all tool-specific temporary variables introduced, and consequently, the native methods summarized).

For a total of 12 programs × 4 analyses = 48 configurations evaluated, QILIN yields the same results as DOOP for all the four metrics. In addition, we have also validated that both produce exactly the same points-to sets for all the variables considered. Thus, QILIN represents a modern framework for supporting precise pointer analyses for large Java programs.

Note that QILIN and DOOP may introduce a few different temporary variables in modeling native methods and certain language constructs. The differences in their points-to facts will not affect the points-to information computed for the variables in a Java method.

5.2 RQ2: Efficiency

Table 5 also compares QILIN with DOOP in terms of the efficiency of *insens*, 1CFA, 2OBJ, and S-2OBJ achieved. The time budget for running an analysis on a program is 12 hours. The analysis time of a program is an average of 3 runs. QILIN currently uses a single-threaded constraint solver while DOOP uses a multi-threaded Datalog engine, Soufflé. According to [3], Soufflé delivers its maximum performance at 4 or 8 threads. While DOOP defaults to 4 threads, we have used 8 threads to enable it to achieve slightly better performance. Note that the analysis time of a program under DOOP is given as the analysis time spent by its Datalog engine only (without including the time spent by its fact generator, which is claimed to be amortizable across a number of analyses applied to the same program).

For the same 12 programs × 4 analyses = 48 configurations evaluated, QILIN outperforms DOOP substantially, with the speedups ranging from 0.9x (for `checkstyle` under 2OBJ) to 6.3x (for `xalan` also under 2OBJ). Note that QILIN is slightly slower than DOOP only under 2OBJ in analyzing `checkstyle`. The overall average speedup achieved by QILIN over DOOP for all the four analyses across the 12 programs is 2.4x. This increases to 2.9x when DOOP switches from 8 to 4 threads and 5.1x when DOOP switches to a single thread.

As QILIN achieves exactly the same precision as DOOP (Section 5.1), its high performance is attributed to our new incremental worklist-based constraint solver, which runs significantly faster than Soufflé [39, 20]. Thus, this work confirms (for the first time) that an imperative framework (implemented in Java) that relies on a well-crafted constraint solver can outperform a declarative counterpart that relies on a (multi-threaded) Datalog engine, despite that QILIN is currently single-threaded and DOOP has been carefully optimized in over one decade. QILIN's high efficiency is expected to provide significant performance benefits for its client applications, such as call graph construction tools [23, 1, 38], bug detection tools [34, 55, 27, 10] and compiler optimization techniques[9, 47].

Table 5 The efficiency and precision of QILIN and DOOP in supporting *insens*, 1CFA, 2OBJ, and S-2OBJ. For all metrics (except speedups of QILIN over DOOP in blue), smaller is better.

Program	Metrics	*insens*		1cfa		2obj		S-2obj	
		DOOP	QILIN	DOOP	QILIN	DOOP	QILIN	DOOP	QILIN
antlr	Time (s)	27	11 (**2.5x**)	61	24 (**2.6x**)	116	55 (**2.1x**)	90	46 (**2.0x**)
	#call-edges	57472	57472	56226	56226	51319	51319	51318	51318
	#fail-casts	1127	1127	930	930	511	511	439	439
	#poly-calls	1987	1987	1933	1933	1643	1643	1642	1642
	#avg-pts	36.498	36.498	32.106	32.106	9.055	9.055	8.945	8.945
bloat	Time (s)	20	11 (**1.8x**)	85	30 (**2.9x**)	1503	788 (**1.9x**)	1450	800 (**1.8x**)
	#call-edges	67856	67856	65689	65689	56837	56837	56836	56836
	#fail-casts	2088	2088	1891	1891	1316	1316	1244	1244
	#poly-calls	2344	2344	2171	2171	1714	1714	1713	1713
	#avg-pts	52.992	52.992	47.391	47.391	15.387	15.387	15.287	15.287
chart	Time (s)	51	19 (**2.7x**)	174	41 (**4.3x**)	411	222 (**1.9x**)	435	370 (**1.2x**)
	#call-edges	86806	86806	84116	84116	72805	72805	72801	72801
	#fail-casts	2563	2563	2207	2207	1348	1348	1183	1183
	#poly-calls	2732	2732	2614	2614	2068	2068	2067	2067
	#avg-pts	64.751	64.751	52.949	52.949	5.796	5.796	5.541	5.541
eclipse	Time (s)	115	37 (**3.1x**)	482	132 (**3.7x**)	8556	4701 (**1.8x**)	8493	4266 (**2.0x**)
	#call-edges	183288	183288	178585	178585	162934	162934	162876	162876
	#fail-casts	5114	5114	4732	4732	3648	3648	3542	3542
	#poly-calls	10738	10738	10455	10455	9773	9773	9718	9718
	#avg-pts	137.322	137.322	62.446	62.446	16.115	16.115	14.944	14.944
fop	Time (s)	29	9 (**3.3x**)	51	17 (**3.1x**)	52	25 (**2.1x**)	48	22 (**2.2x**)
	#call-edges	40558	40558	39285	39285	34424	34424	34424	34424
	#fail-casts	914	914	710	710	396	396	315	315
	#poly-calls	1223	1223	1156	1156	842	842	842	842
	#avg-pts	25.526	25.526	20.469	20.469	4.399	4.399	4.262	4.262
luindex	Time (s)	12	9 (**1.3x**)	28	15 (**1.9x**)	38	25 (**1.6x**)	34	23 (**1.5x**)
	#call-edges	39809	39809	38529	38529	33643	33643	33642	33642
	#fail-casts	923	923	727	727	396	396	324	324
	#poly-calls	1294	1294	1228	1228	935	935	934	934
	#avg-pts	20.807	20.807	16.290	16.290	4.480	4.480	4.325	4.325
lusearch	Time (s)	13	9 (**1.4x**)	31	17 (**1.9x**)	73	37 (**2.0x**)	67	36 (**1.9x**)
	#call-edges	43153	43153	41841	41841	36525	36525	36524	36524
	#fail-casts	1035	1035	831	831	411	411	332	332
	#poly-calls	1505	1505	1432	1432	1133	1133	1132	1132
	#avg-pts	22.418	22.418	17.625	17.625	4.461	4.461	4.299	4.299
pmd	Time (s)	35	14 (**2.6x**)	103	30 (**3.5x**)	104	53 (**2.0x**)	94	56 (**1.7x**)
	#call-edges	69713	69713	67899	67899	60030	60030	60029	60029
	#fail-casts	2273	2273	2026	2026	1416	1416	1333	1333
	#poly-calls	2989	2989	2871	2871	2390	2390	2389	2389
	#avg-pts	37.331	37.331	31.764	31.764	6.036	6.036	5.947	5.947
xalan	Time (s)	39	11 (**3.6x**)	68	22 (**3.1x**)	4046	638 (**6.3x**)	823	405 (**2.0x**)
	#call-edges	54147	54147	52657	52657	46856	46856	46855	46855
	#fail-casts	1305	1305	1058	1058	601	601	516	516
	#poly-calls	2101	2101	2010	2010	1657	1657	1656	1656
	#avg-pts	29.968	29.968	24.529	24.529	6.037	6.037	5.924	5.924
checkstyle	Time (s)	64	16 (**4.0x**)	146	36 (**4.1x**)	6308	7148 (**0.9x**)	5338	5535 (**1.0x**)
	#call-edges	80291	80291	77881	77881	67285	67285	67276	67276
	#fail-casts	1941	1941	1680	1680	1117	1117	1023	1023
	#poly-calls	2778	2778	2655	2655	2241	2241	2234	2234
	#avg-pts	47.925	47.925	39.536	39.536	8.048	8.048	7.706	7.706
JPC	Time (s)	32	21 (**1.5x**)	127	40 (**3.2x**)	211	132 (**1.6x**)	264	192 (**1.4x**)
	#call-edges	95055	95055	91661	91661	81465	81465	81429	81429
	#fail-casts	2254	2254	1894	1894	1357	1357	1208	1208
	#poly-calls	4960	4960	4840	4840	4282	4282	4275	4275
	#avg-pts	45.533	45.533	32.175	32.175	6.045	6.045	5.841	5.841
findbugs	Time (s)	54	20 (**2.8x**)	198	48 (**4.1x**)	3887	1644 (**2.4x**)	3856	1593 (**2.4x**)
	#call-edges	106065	106065	102352	102352	88107	88107	88107	88107
	#fail-casts	3457	3457	3000	3000	2058	2058	1968	1968
	#poly-calls	4534	4534	4308	4308	3679	3679	3679	3679
	#avg-pts	64.510	64.510	53.842	53.842	9.102	9.102	9.052	9.052

Table 6 Human effort required in integrating fine-grained context-sensitive analysis into QILIN.

Pointer Analysis	Source Code (#LOC)	Supporting Code (#LOC)
BEAN [49]	355	41
MAHJONG [48]	666	51
Data-Driven [19]	1062	39
ZIPPER [25]	1474	35
Context-Tunneling [17]	1151	29
EAGLE [30]	357	75
TURNER [14]	769	75
CONCH [15]	1230	55

5.3 RQ3: Modularity

Table 7 The efficiency and precision of E-2OBJ, T-2OBJ, and Z-2OBJ for performing kOBJ under fine-grained context-sensitivity enabled by EAGLE [30], TURNER [14], and ZIPPER [25], respectively. The speedups of each analysis (in blue) is computed with 2OBJ (shown in Table 5) as the baseline.

Metrics	Program	E-2obj	T-2obj	Z-2obj	Program	E-2obj	T-2obj	Z-2obj
Time (s)	antlr	38 (**1.4x**)	18 (**3.1x**)	31 (**1.8x**)	lusearch	27 (**1.4x**)	19 (**1.9x**)	19 (**1.9x**)
#call-edges		51319	51319	51505		36525	36525	36720
#fail-casts		511	511	532		411	411	441
#poly-calls		1643	1643	1666		1133	1133	1162
#avg-pts		9.055	9.067	9.492		4.461	4.473	5.071
Time (s)	bloat	577 (**1.4x**)	302 (**2.6x**)	663 (**1.2x**)	pmd	38 (**1.4x**)	25 (**2.1x**)	31 (**1.7x**)
#call-edges		56837	56837	57059		60030	60030	60180
#fail-casts		1316	1316	1339		1416	1416	1447
#poly-calls		1714	1714	1746		2390	2390	2412
#avg-pts		15.387	15.403	16.381		6.036	6.045	6.441
Time (s)	chart	165 (**1.3x**)	119 (**1.9x**)	45 (**4.9x**)	xalan	366 (**1.7x**)	297 (**2.1x**)	285 (**2.2x**)
#call-edges		72805	72805	73243		46856	46856	47005
#fail-casts		1348	1348	1395		601	601	618
#poly-calls		2068	2068	2094		1657	1657	1680
#avg-pts		5.796	5.809	6.474		6.037	6.055	6.550
Time (s)	eclipse	2929 (**1.6x**)	1904 (**2.5x**)	2000 (**2.4x**)	checkstyle	3776 (**1.9x**)	2813 (**2.5x**)	1882 (**3.8x**)
#call-edges		162934	162934	163176		67285	67285	67511
#fail-casts		3648	3648	3707		1117	1117	1144
#poly-calls		9773	9773	9834		2241	2241	2278
#avg-pts		16.115	16.305	16.413		8.048	8.152	9.257
Time (s)	fop	17 (**1.5x**)	12 (**2.1x**)	13 (**1.9x**)	JPC	100 (**1.3x**)	75 (**1.8x**)	54 (**2.4x**)
#call-edges		34424	34424	34615		81465	81465	81741
#fail-casts		396	396	421		1357	1357	1393
#poly-calls		842	842	868		4282	4282	4337
#avg-pts		4.399	4.416	4.977		6.045	6.062	6.514
Time (s)	luindex	18 (**1.4x**)	11 (**2.3x**)	14 (**1.8x**)	findbugs	925 (**1.8x**)	179 (**9.2x**)	160 (**10.3x**)
#call-edges		33643	33643	33833		88107	88107	88172
#fail-casts		396	396	422		2058	2058	2091
#poly-calls		935	935	960		3679	3679	3687
#avg-pts		4.480	4.494	5.065		9.102	9.139	9.300

QILIN supports a variety of context-sensitive pointer analyses that can all be specified modularly as variations on a common code base with their context-sensitivity parameterized by `Cons`, `Sel`, and `HeapAbs`. We use #LOC, the number of LOC required in integrating a pointer analysis algorithm into QILIN, to measure the modularity of our framework in supporting the design and implementation of new algorithms. While #LOC is not equivalent

to the amount of engineering efforts involved, a small #LOC needed indicates that our framework is highly modular. For the four traditional method-level analyses, *insens*, kCFA, kOBJ and S-kOBJ, listed in Table 4 and evaluated above, their parameterization requires 15, 43, 40 and 61 LOC, respectively, totaling only 98 LOC with the commonalities factored out.

In addition, QILIN also accommodates well a range of recently proposed fine-grained context-sensitive pointer analyses [49, 48, 19, 17, 25, 30, 14], as demonstrated in Table 6. For each analysis, the second column lists the number of LOC required for defining the context-sensitivity proposed (in the form of a pre-analysis), and the third column gives the number of LOC required for parameterizing it in QILIN (requiring only an average of 50 LOC each). We have integrated all these seven analyses into QILIN except the machine learning phases used in DATA-DRIVEN [19] and Context-Tunneling [17].

5.4 RQ4: Fine-Grained Context-Sensitivity

QILIN is expected to represent a common framework in which different pointer analysis algorithms can be designed and evaluated effectively. Table 7 compares the efficiency and precision of E-2OBJ, T-2OBJ, and Z-2OBJ (the three fine-grained variations of 2OBJ listed in Table 4) enabled by EAGLE [30], TURNER [14], and ZIPPER [25], respectively, with all the parameterization efforts given in Table 6 (in terms of #LOC added). The speedups of each of the three analyses is computed with 2OBJ (shown in Table 5) as the baseline.

Precision-wise, our results are consistent with those reported earlier in [30, 25, 14]. Specifically, E-2OBJ always preserves the precision of 2OBJ in theory, T-2OBJ preserves the precision of *#call-edges*, *#fail-cast*, and *#poly-calls* but not *#avg-pts* in practice, and finally, Z-2OBJ loses precision by design in general as it has caused *#call-edges*, *#fail-cast*, *#poly-calls* and *#avg-pts* to increase by 3.6%, 1.6%, 0.4% and 8.7%, respectively.

Efficiency-wise, our results are also consistent with those reported earlier in [30, 25, 14] in the sense that E-2OBJ, T-2OBJ, and Z-2OBJ are faster than 2OBJ. Specifically, ZIPPER (the least precise) achieves the highest speedups, ranging from 1.2x (for `bloat`) to 10.3x (for `findbugs`) with an average of 3.0x, TURNER achieves slightly lower speedups, ranging from 1.8x (for `JPC`) to 9.2x (for `findbugs`) with an average of 2.8x, and finally, EAGLE (the most precise) achieves the lowest speedups, ranging from 1.3x (for `JPC`) to 1.9x (for `checkstyle`) with an average of 1.5x. However, the relative speedups of E-2OBJ, T-2OBJ, and Z-2OBJ over 2OBJ reported here are not expected to be exactly the same as those reported in [30, 14] due to different experimental settings used (for the purposes of validating different design hypotheses). One difference is particularly noteworthy: E-2OBJ and Z-2OBJ are compared by parameterizing E-kOBJ and Z-kOBJ as in Table 4 in [14], but E-kOBJ as [Eq. (3), Eq. (10), Eq. (11)] and Z-kOBJ as [Eq. (3), Eq. (9), Eq. (11)] instead in [14, 30]. Specifically, the objects that are instantiated from `StringBuilder` and `StringBuffer` as well as `Throwable` (including its subtypes) are merged per dynamic type and then analyzed context-insensitively in [14, 25] but not in [30]. This again highlights the significance for the research community to share QILIN as a common open-source framework to design and evaluate different fine-grained analyses in future work.

6 Related Work

We review the past work on context-sensitive pointer analyses for Java by focusing on representative open-source frameworks developed and the recent research trend on exploring fine-grained context-sensitivity, by placing QILIN again in its research context.

Pointer Analysis Frameworks. Existing frameworks, which were originally designed and implemented to support method-level context-sensitivity, fall into three categories: (1) imperative, e.g., Spark [23] and Wala [16] (implemented in Java), (2) declarative, e.g., Doop [8] (coded in Datalog on top of a Datalog engine, e.g., Soufflé [20, 39]), and (3) hybrid, e.g., Jchord [34] and Paddle [24] (with the core of a pointer analysis algorithm performed in Datalog declaratively but the rest coded in Java imperatively). In contrast, Qilin is a new framework designed to support variable-level context-sensitivity (by subsuming existing traditional frameworks as a special case since all the variables/objects in a method can only be analyzed traditionally by using exactly the same context abstraction).

In the past decade or so, Doop has been the state of the art for supporting traditional pointer analyses with method-level context-sensitivity. As a fully-declarative framework, Doop is highly scalable, enabling complex and precise context-sensitive pointer analyses to be developed efficiently in the past [8, 42]. Whether an imperative alternative can outperform Doop in terms of efficiency while achieving the same precision remains to be unknown for years in the pointer analysis community. In this paper, we show that Qilin, as an imperative framework, can outperform Doop substantially while also allowing all traditional pointer analyses to be specified precisely and modularly as in Doop.

Fine-Grained Context-Selectivity. To scale context-sensitive analyses further for large codebases, how to explore a significantly larger space of efficiency/precision tradeoffs by moving from method-level to fine-grained context-sensitivity has received increasing interest. In the past few years, method-level context-sensitivity has been made (1) selective (by analyzing only a subset of methods context-sensitively via exploiting user-supplied hints [43], machine learning [19], and pattern-matching [25], and (2) selective (by analyzing only a subset of variable/objects context-sensitively via exploiting CFL (Context-Free Language) reachability [28, 30, 14, 29]. In the near future, we envisage to see more pointer analyses with variable-level context-sensitivity to be developed.

However, existing pointer analysis frameworks [23, 16, 8, 34, 24] were all designed for supporting method-level context-sensitivity. As described in Section 1, we have made significant efforts in extending Doop to support fine-grained context-sensitivity, but to no avail. We see two limitations from our preliminary investigation: First, the number of rules for supporting fine-grained analyses increases drastically relative to the Doop baseline since a fine-grained analysis relies on more configurable parameters (in addition to the context length k). Second, the performance of the extended Doop version is rather disappointing due to possibly poor join orders selected by its underlying Datalog engine [39, 20] used.

In this paper, we have designed and implemented Qilin on top of Spark [23] for supporting fine-grained context-sensitivity. In our previous work [28, 30, 14, 29], we have also introduced a few in-house implementations (which can be seen as some precursors of Qilin) for supporting only partial context-sensitivity (under which variables/objects and methods can be analyzed either context-sensitively or context-insensitively). These implementations are designed to support specific pointer analysis techniques with different design choices and settings for handling different language features, which limit them from being used as a general-purpose framework. To the best of our knowledge, Qilin represents the first framework that supports all such fine-grained analyses precisely, efficiently and modularly.

Iterative Constraint Solving via Difference Propagation. Many program analyses, such as pointer analysis, exploit the idea of difference propagation [12, 35, 23] when resolving their constraints towards a fixed-point solution efficiently. For example, Sridharan et al. [45] present

a difference-propagation-based pointer analysis algorithm for object-oriented programs. In their algorithm, every time when an edge $x \xrightarrow{\text{assign}} y$ needs to be handled, their algorithm needs to compute $\delta = \texttt{PTS}(x) - \texttt{PTS}(y)$ and then propagates δ to $\texttt{PTS}(y)$, which can be highly expensive. In contrast, QILIN's incremental worklist-based algorithm (Algorithm 1), which is is extended from SPARK [23], computes only $\delta_{\text{new}} = \texttt{PTS}(x)_{\text{new}} - \texttt{PTS}(y)_{\text{new}}$ and then propagates δ_{new} to $\texttt{PTS}(y)_{\text{new}}$. As a result, our constraint solver is more efficient. In addition, the semi-naïve evaluation, an efficient evaluation strategy used by many existing Datalog engines [20, 31], refines the naïve (chaotic iteration) strategy to avoid redundant work by exploiting also the idea of difference propagation. The speedups achieved by QILIN over DOOP are attributed to our incremental worklist algorithm, which may exhibit better join orders than the ones automatically selected by DOOP's underlying Datalog engine [20].

7 Conclusion and Future work

We have introduced QILIN as the first open-source framework (to be released soon) for supporting fine-grained context-sensitive pointer analyses (including the traditional ones as special cases) for Java, precisely, efficiently and modularly. Developing such a production-quality framework involves a lot of technical and engineering efforts. We will maintain and grow this open-source project actively on GitHub to support further research on pointer analysis and a variety of other static analyses for Java (and possibly other object-oriented programming languages). Several immediate future research/engineering activities that can be carried out in QILIN include (1) parallelizing its constraint solver to lift its performance further, (2) covering more native methods and Java reflection APIs, (3) supporting more Java features in JDK8 and above, and (4) experimenting with the design and implementation of novel variable-level context-sensitive pointer analysis algorithms. Finally, as an open-source project, QILIN is also expected to be driven by community contribution.

References

1. Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *ECOOP 2012 – Object-Oriented Programming*, pages 688–712, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
2. Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.
3. Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting Doop to Soufflé: A tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 25–30, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3088515.3088522.
4. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, New York, NY, USA, 2014. Association for Computing Machinery.
5. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

6 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, Honolulu, HI, USA, 2011. IEEE.

7 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

8 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery.

9 Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 416–425, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1168857.1168908.

10 Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. SMOKE: Scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82, New York, NY, USA, 2019. IEEE. doi:10.1109/ICSE.2019.00025.

11 Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3092703.3092729.

12 Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *European Symposium on Programming*, pages 90–104. Springer, 1998.

13 Neville Grech and Yannis Smaragdakis. P/Taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017. doi:10.1145/3133926.

14 Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021)*, pages 18:1–18:31, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

15 Dongjie He, Jingbo Lu, and Jingling Xue. Context debloating for object-sensitive pointer analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 79–91. IEEE, 2021.

16 IBM. WALA: T.J. Watson Libraries for Analysis, 2020. URL: http://wala.sourceforge.net/.

17 Minseok Jeon, Sehun Jeong, and Hakjoo Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

18 Minseok Jeon, Myungho Lee, and Hakjoo Oh. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

19 Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):100, 2017.

20 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

21 Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 774–784, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3338906.3338936.

22 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–434, New York, NY, USA, 2013. Association for Computing Machinery.

23 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

24 Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), October 2008. doi:10.1145/1391984.1391987.

25 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

26 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 129–140, New York, NY, USA, 2018. Association for Computing Machinery.

27 Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 359–373, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192390.

28 Jingbo Lu, Dongjie He, and Jingling Xue. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–46, 2021.

29 Jingbo Lu, Dongjie He, and Jingling Xue. Selective context-sensitivity for k-CFA with CFL-reachability. In *International Static Analysis Symposium*, pages 261–285. Springer, 2021.

30 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

31 Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From Datalog to flix: A declarative language for fixed points on lattices. *ACM SIGPLAN Notices*, 51(6):194–208, 2016.

32 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. Association for Computing Machinery.

33 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

34 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. Association for Computing Machinery.

35 David J Pearce, Paul HJ Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 3–12. IEEE, 2003.

36 Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, pages 361–369, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3196321.3197546.

37 Jyoti Prakash, Abhishek Tiwari, and Christian Hammer. Effects of program representation on pointer analyses – an empirical study. *Fundamental Approaches to Software Engineering*, 12649:240, 2021.

38 Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for Java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 474–486, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2950290.2950312.

39 Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892208.2892226.

40 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

41 Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

42 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, New York, NY, USA, 2011. Association for Computing Machinery.

43 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–495, New York, NY, USA, 2014. Association for Computing Machinery.

44 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

45 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, Berlin, Heidelberg, 2013.

46 Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, New York, NY, USA, 2007. Association for Computing Machinery.

47 Yulei Sui, Yue Li, and Jingling Xue. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, New York, NY, USA, 2013. IEEE. doi:10.1109/CGO.2013.6494978.

48 T. Tan, Y. Li and J. Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–291, New York, NY, USA, 2017. Association for Computing Machinery.

49 Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*, pages 489–510, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

50 Rei Thiessen and Ondřej Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–277, New York, NY, USA, 2017. Association for Computing Machinery.

51 David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. Past-sensitive pointer analysis for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 197–208, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368089.3409698.

52 David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 350–360, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180251.

53 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., USA, 2010.

54 Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

55 Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337, New York, NY, USA, 2018. IEEE. doi:10.1145/3180155.3180178.

NWGraph: A Library of Generic Graph Algorithms and Data Structures in C++20

Andrew Lumsdaine
University of Washington, Seattle, WA, USA
Pacific Northwest National Laboratory, Richland, WA, USA
TileDB, Inc., Cambridge, MA, USA

Luke D'Alessandro
Indiana University, Bloomington, IN, USA

Kevin Deweese
Cadence Design Systems, San Jose, CA, USA

Jesun Firoz
Pacific Northwest National Laboratory, Richland, WA, USA

Xu Tony Liu
University of Washington, Seattle, WA, USA

Scott McMillan
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA

John Phillip Ratzloff
SAS Institute, Cary, NC, USA

Marcin Zalewski
NVIDIA, Seattle, WA, USA

Abstract

The C++ Standard Library is a valuable collection of generic algorithms and data structures that improves the usability and reliability of C++ software. Graph algorithms and data structures are notably absent from the standard library, and previous attempts to fill this gap have not gained widespread adoption. In this paper we show that the richness of graph algorithms and data structures can in fact be captured by straightforward composition of existing C++ mechanisms. Generic programming is algorithm-oriented. Accordingly, we apply a systematic approach to analyzing a broad set of graph algorithms, "lift" unnecessary constraints from them, and organize the resulting set of minimal common *type requirements*, i.e., concepts, for defining their interfaces. By using the newly available *ranges* and *concepts* in C++20, the type requirements for generic graph algorithms can be succinctly expressed. The generic algorithms and data structures resulting from our analysis are realized in NWGraph, a modern, composable, and extensible C++ library.

2012 ACM Subject Classification Software and its engineering → Software libraries and repositories; Mathematics of computing → Graph algorithms

Keywords and phrases Graph library, generic programming, graph algorithms

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.31

Supplementary Material *Software (Source Code)*: `https://github.com/pnnl/NWGraph`
archived at `swh:1:dir:50db7d4a73652c1073d8141a1e3d83896b0ca3b0`

Funding This work was partially supported by the High Performance Data Analytics (HPDA) program and the Segmented Global Address Space (SGAS) LDRD under the Data Model Convergence (DMC) initiative at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. This research was partially supported by NSF Awards OAC-1716828 and OAC-2126266. This material is also based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM22-0432].

1 Introduction

Graphs are powerful mathematical tools for reasoning about the relationships between given entities, focusing on the *structure and characteristics of the relationships*, independent of what the entities and the relationships actually are. Consequently, results from graph theory

© Andrew Lumsdaine, Luke D'Alessandro, Kevin Deweese, Jesun Firoz, Xu Tony Liu, Scott McMillan, John Phillip Ratzloff, Marcin Zalewski, and Carnegie Mellon University;
licensed under Creative Commons License CC-BY 4.0

can be applied to any actual sets of data for which relationships between elements can be established. Internet packet routing, molecular biology, electronic design automation, social network analysis, and search engines are just some of the problem areas where graph theory is regularly applied. The general applicability we find in graph theory – the *genericity*, if you will – is a goal for software libraries as well as mathematical theories; graph algorithms and data structures (collectively, "graphs") would seem to be ideally suited for software reuse.

Realizing a truly generic library for graphs has significant challenges in practice. Graphs in theory are useful because they are abstract, but, in practice, they have to be made concrete when used to solve an actual problem. That is, graphs in practice do not manifest themselves in the abstract form to which theory and abstract algorithms are applied. Rather, they are often encoded in some domain-specific form or are latent in problem-related data. And even if a domain programmer constructs a graph from their data, the domain-specific graph data structure might not be compatible with the API of a given graph library.

The celebrated Standard Template Library (STL), now part of the C++ standard library, addressed this problem for fundamental algorithms and abstract containers of data elements [30]. With the STL, *generic programming* emerged as a software-development sub-discipline that focused on creating frameworks of reusable and composable libraries. Fundamental to the philosophy of generic programming is that algorithms should be able to be composed with arbitrary *types*, notably types that may have been developed completely independently of the library. To achieve this goal, generic algorithms are specified and written in terms of *abstract properties of types*; a generic algorithm can be composed with any type meeting the properties that it depends on. Philosophically, generic programming goes hand-in-glove with the abstraction process inherent in graph theory. Graphs are abstract models of entities in relationship – a graph algorithm should be able to operate directly on the entities and relationships in a programmer's data.

It is not just the philosophy of generic programming from the STL that can be leveraged to develop a generic graph library. In fact, an important principle upon which our work is based is that **the standard library already contains sufficient capability to support graph algorithms and data structures.** The *type requirements for generic graph algorithms* can be expressed using existing type requirement machinery for standard library algorithms, and useful and efficient graph algorithms can be implemented based on these requirements.

We apply this principle to develop NWGraph, a generic library of algorithms for graph computation that are independent of any particular data structure (in particular, independent of any particular graph data structure). Following current generic library practice, NWGraph algorithms are organized around a minimal set of common requirements for their input types (these requirements are formalized in the form of C++20 *concepts*).

The foundation of this paper is a requirements analysis from which we derive a uniform set of type requirements for graph algorithms; those requirements subsequently reified as C++ concepts. Based on this foundation, we construct the primary components of NWGraph: **algorithms**, defined and and implemented using our concepts; **adaptors**, for converting one representation of a graph into another and for enabling structured traversals, and **data structures** that model our foundational requirements.

NWGraph contains the following innovations:

- A concept taxonomy (expressed using C++20 concepts) for specifying graph algorithm requirements;
- Characterization of graphs using standard library concepts (as a random access range of forward ranges);
- A rich set of range adaptors for accessing and traversing graphs;
- An API designed to fully support modern idiomatic C++;

- An efficient and fully parallelized implementation, using C++ execution policies and Intel® Threading Building Blocks; and
- Application of the generic programming process to minimize requirements on algorithm input types (thereby enlarging the scope of composability).

In the following sections we first provide some basic background and terminology that we will be using to discuss graph algorithms (§2) as well as a bit more detail on generic programming (§3). Next, we analyze the domain of graph algorithms with respect to common requirements and present the fundamental concepts in NWGraph (§4). We then present an overview of the primary components of NWGraph in addition to its concepts: its algorithms (§5), adaptors (§6), and data structures (§7). We include abstraction penalty experiments, evaluate the performance of our library in comparison with other well-known graph libraries, and conduct a strong scaling study of the parallel performance of NWGraph (§8). Finally, we provide a high-level feature comparison of NWGraph with other extant graph libraries (§9) and conclude with some of our observations and experiences in developing NWGraph (§10). NWGraph is hosted at `https://github.com/pnnl/NWGraph`.

2 Graph Background

We define a *graph* G as comprising two finite sets, $G = \{V, E\}$, where the set $V = \{v_0, v_1, \ldots v_{n-1}\}$ is a set of entities of interest, "vertices" or "nodes," and $E = \{e_0, e_1, \ldots e_{m-1}\}$ is a set of pairs of entities from V, "edges" or "links." Edges may be ordered or unordered; a graph defined with ordered edges is said to be *directed*; a graph defined with unordered edges is said to be *undirected*.

▶ Remark. Understanding graphs is necessary to develop requirements for algorithms. However, it should be noted that we don't derive those requirements from the graph model, but *instead from the algorithms*. This is a key distinction between generic programming and, say, Object-Oriented (OO) requirements analysis.

2.1 Representing Graphs

To define algorithms on graphs and to be able to reason about those algorithms, we need to define some representations for graphs; not much can be done computationally with abstract sets of vertices and edges. The specific characteristics of these representations are what we use to express algorithms (still abstractly) but when those algorithms are implemented as generic library functions, those characteristics will in turn become the basis for the library's interfaces (represented in our case as C++ concepts).

One of the fundamental operations in graph algorithms is a *traversal*. That is, given a vertex u, we would like to find the *neighbors* of u, i.e., all vertices v such that the edge (u, v) is in the graph. Then, for each of those edges, we would like to find their neighbors, and so on. The representation that we can define to make this efficient is an *adjacency list*.

Given a graph $G = (V, E)$, we can define an adjacency-list representation in the following way. Assign to each element of V a unique index from the range $[0, |V|)$ and denote the vertex identified with index i as $V[i]$. We can now define a new graph with the same structure as G, but in terms of the indices in $[0, |V|)$, rather than with the elements in V. Let the *index graph of G* be the graph $G' = (V', E')$, where $V' = [0, |V|)$ and E' consists of $|E|$ pairs of indices from V, such that a pair (i, j) is in E' if and only if $(V[i], V[j])$ is in E. Which is all to say, the index graph of G is the graph we get by replacing all elements of G with their corresponding indices.

We make the following definition: An *adjacency list* of an index graph $G = (V, E)$ is an array $Adj(G)$ of size $|V|$ (the array is indexed from 0 to $|V|-1$) with the following properties:

- $Adj(G)$ is a container of $|V|$ containers, one container for each vertex in V, and
- The container $Adj(G)[u]$ contains all vertices v for which there is an edge $(u, v) \in E$.

This structure, an adjacency list of an index graph, or an index adjacency list, is the fundamental structure used by almost all graph algorithms.

▶ Remark 1. Although the standard term for this kind of abstraction is "adjacency list", and although it is often drawn schematically with linked lists as elements, it is not necessary that this abstraction be implemented as an actual linked list. In fact, other representations (such as compressed sparse row storage) are significantly more efficient, as we show in Section 8.3. What is important is that the items that are stored, vertex indices, can be used to index into the adjacency list to obtain other lists of neighbors.

▶ Remark 2. An adjacency list does not store edges per se, rather it stores lists of reachable neighbors. Therefore, though it can represent a directed or undirected graph, an adjacency list is structurally neither inherently directed nor undirected. That is, given vertex u, the container $Adj(G)[u]$ contains the vertex v if the edge (u, v) is contained in E, i.e., for a directed graph with edge (u, v) in E, $Adj(G)[u]$ will contain v. For an undirected graph with edge (u, v) contained in E, $Adj(G)[u]$ will contain v **and** $Adj(G)[v]$ will contain u. Thus, directedness of the original graph is made manifest in the *values* stored in the adjacency list, not in its structure.

3 Generic Programming

Generic programming is a software development paradigm inspired by the organizational principles of mathematics [31]. That is, a generic library comprises a framework of algorithms in a problem domain, based on a systematic organization of common type requirements for those algorithms. The type requirements themselves, specified as *concepts*, are part of the library as well, and provide the interface that enables composition of library components with other, independently-developed, components. Concrete types that meet the requirements of a concept are said to *model* that concept. As an example, the `iterator` concept taxonomy was the foundation upon which the STL was organized [21, 30].

Generic algorithms (that is, algorithms in a generic library) are designed so that the requirements they impose on types are as minimal as possible without compromising efficiency, thus enabling the widest scope of potential composition, and therefore, reuse. Generic algorithms are derived from concrete ones, which are gradually made more generic by removing ("lifting") unnecessary requirements. This process continues as long as instantiation of the generic algorithm with concrete types remains as efficient as the equivalent concrete algorithm would have been.

3.1 Lifting

The first (and major) phase of the generic programming process is sometimes known as "lifting" where we create generic algorithms through a process of successive generalization. That is, the process is

1. Study the concrete implementation of an algorithm;
2. Lift away unnecessary requirements to produce a more abstract algorithm;
3. Repeat the lifting process until we have obtained a generic algorithm that is as general as possible but that still instantiates to efficient concrete implementations; and
4. Catalog remaining requirements and organize them into concepts.

Listing 1 Concrete implementation for summing elements of an array.

```
int sum(int *array, int n) {
  int s = 0;
  for (int i = 0; i < n; ++i) {
    s = s + array[i]; }
  return s; }
```

Listing 2 Lifted implementation of sum, where traversal through the container and element access has been abstracted through the use of iterators and addition has been further lifted with the introduction of the operator parameter `op`.

```
template <class Iter, class T, class Op>
T accumulate(Iter first, Iter last,  T s, Op op) {
  while (first != last) {
    s = op(s, *first++); }
  return s; }
```

Listings 1–2 show two concrete implementations of a `sum` algorithm. The first steps through an array of integers, indexing into the array at each step and summing the resulting value into `s`. Instead of an array, any eligible container (for example, linked list) can store the values.

The authors of the STL realized the commonality of traversal and element access across most basic computer science algorithms. The requirements for traversal and access were generalized and unified into a hierarchy of type requirements for iterators [30].

An iterator-based algorithm for accumulating elements in a container is shown in Listing 2. Note that this single parameterized algorithm replaces the `sum` algorithm shown in Listing 1 (and more). The process of summation has further been generalized by the introduction of a function object `op` as a parameter to the function.

3.2 Specialization

In generic programming, the dual to lifting is *specialization*. That is, once an algorithm is lifted and made generic, it is specialized through composition with a concrete data type to realize a concrete implementation of the algorithm. Listing 3 shows two example usages of the generic `accumulate`, composing it with an array as well as a linked list from the STL.

Now, there is a crucial requirement that is part of specialization. In generic programming, we don't just require that when we have a lifted algorithm that we can compose it with the data types that we lifted from. In addition to that basic requirement, we also require that *there is zero abstraction penalty.* That is, the specialized generic algorithm should provide

Listing 3 Specializations of the generic `accumulate` algorithms shown in Listing 2. The `accumulate` algorithm is composed with an integer array (left) and accumulate is composed with a linked list (right).

```
int* array = new int [10];
int result =
    accumulate(array, array + 10,
     0, std::plus<int>());
```

```
std::forward_list<double> ptr;
double result = accumulate(ptr,
    nullptr, 0.0,
    std::times<double>());
```

Listing 4 Skeleton of the requirements for a C++ `input_iterator`.

```
1  template <class I>
2  concept input_iterator = requires(I i) {
3    typename std::iter_value_t<I>;
4    typename std::iter_reference_t<I>;
5    { *i } -> std::same_as<std::iter_reference_t<I>>;
6    { ++i } -> std::same_as<I &>;
7    i++;};
```

exactly the same performance as the concrete algorithm from which it was lifted, when composed with the original types that were lifted. With modern compilers and libraries, this requirement is actually met, and is one of the reasons that libraries such as the C++ standard library have been so successful in practice.

3.3 Concepts in C++20

In generic programming, concepts consist of valid expressions and associated types, which define a family of allowable types admissable for composition with generic algorithms. Introduced as a language feature for C++20, concepts constrain the set of types that can be substituted for class and function template arguments. This development has been instrumental in the notable development of the ranges algorithm library taxonomy, serving as the link between generic algorithm interface and implementation [23].

A C++20 `concept` definition declares a set of requirements on types. There are four types of requirements:

- A simple requirement consists of an arbitrary expression statement. The requirement is that the expression is valid.
- A type requirement consists of the keyword `typename` followed by a type name, optionally qualified. The requirement is that the named type exists.
- A compound requirement specifies a conjunction of arbitrary constraints such as expression constraint, exception constraint, and type constraint, etc.
- A nested requirement consists of another requires-clause, terminated with a semicolon. This is used to introduce predicate constraints expressed in terms of other named concepts applied to the local parameters.

Listing 4 shows the skeleton of the C++ concept definition for `input_iterator`. As hinted in our example, this concept specifies that an `input_iterator` can be de-referenced with `operator*` (line 5) and incremented with `operator++` (lines 6 and 7). Additionally, the concept specifies two associated types: `std::iter_value_t<I>` and `std::iter_reference_t<I>`. Line 5 also indicates that the expression `*i` returns the same type as `std::iter_reference_t<I>`. Again, this example is abbreviated for purposes of illustration. A complete description of the C++20 standard library concepts (including the iterator hierarchy) can be found online at `https://en.cppreference.com/w/cpp/concepts`.

3.4 Ranges in C++20

The new C++20 Ranges library [23] generalizes iterators and containers in C++. Ranges provide a way to make STL algorithms *composable* and improve the readability and writability of C++ code. Ranges consist of a pair of begin and end iterators, which are not required to be the same type. An example of using `ranges` is:

```
std::vector<int> v { /* ... */ }
std::min_element(v.begin(), v.end());//iterator API
std::ranges::min_element(v);         //ranges API
```

In the first case, the generic `min_element` function is called with an iterator pair (`begin` and `end` of the container `v`). In the second case, `min_element` function is called directly with `v` as the parameter, as a `std::vector` is a range (specifically, it satisfies the requirements for the `random_access_range` concept.

C++20 ranges are defined in terms of C++20 concepts. A `std::range` itself is a very straightforward concept:

```
template <class T>
concept range = requires(T& t) {
  ranges::begin(t);
  ranges::end(t); };
```

It has two valid expressions: `begin` and `end`. The `std::input_range`, which abstracts containers that have forward iterators, is thus defined:

```
template<class T>
concept input_range = ranges::range<T>
    &&  std::input_iterator<ranges::iterator_t<T>>;
```

This definition states that an `input_range` is a `range` and that the iterator type associated with that range meets the requirements of the `std::input_iterator` concept.

Related to our development of graph concepts, two range concepts of particular relevance include `ranges::forward_range`, which allows iteration over a collection from beginning to end multiple times (as opposed to an input iterator which is only guaranteed to be able to iterate over a collection once) and `ranges::random_access_range`, which further allows indexing into a collection with `operator[]` in constant time.

4 Generic Graph Algorithms

In this section we analyze the requirements for graph algorithms in order to derive generic graph algorithms. NWGraph realizes these generic algorithms as function templates, and realizes the type requirements as C++20 `concepts`. Our process centers on defining type requirements at the interfaces to algorithms based on what the algorithms actually require of their types, rather than starting with graph types and building algorithms to those types.

4.1 Algorithm Requirements

Algorithms in the STL operate over containers. The concepts defined for the STL have to do with mechanisms for traversing a container and accessing the data therein. Since graphs in some sense are also containers of data, we can reuse the mechanisms from the STL for traversing graphs and accessing graph data, to the extent that makes sense. However, graphs are *structured data* and graph algorithms traverse that structure in various ways. Accordingly, our graph concepts must support structured traversal of graphs.

Most (but not all) graph algorithms traverse a graph vertex to vertex by following the edges that connect vertices. For implementing such algorithms, it is assumed that a graph $G = \{V, E\}$ is represented with an adjacency-list structure[1] as defined in Section 2.1.

[1] The reader is reminded that although the term of art is "adjacency list," containers other than lists can be used to store neighbor information.

BFS(G, s)
```
 1  for each vertex u ∈ V(G)
 2       color[u] ← WHITE
 3  color[s] ← GRAY
 4  Q ← ∅
 5  ENQUEUE(Q, s)
 6  while Q ≠ ∅
 7       u ← DEQUEUE(Q)
 8       for each v ∈ Adj(G)[u]
 9            if color[v] == WHITE
10                 color[v] ← GRAY
11                 ENQUEUE(Q, v)
12       color[u] ← BLACK
```

```
void bfs(const Graph& G, int s) { ...
  for (int u = 0; u < size(G); ++u)
    color[u] = WHITE;
  color[s] = GREY;
  std::queue<int> Q;
  Q.push(s);
  while (!Q.empty()) {
    auto u = Q.front();    Q.pop();
    for (auto&& v : G[u]) {
      if (color[v] == WHITE) {
        color[v] == GREY;
        Q.push(v);            }}
    color[u] = BLACK;         }}
```

Figure 1 Pseudocode and C++ implementation of breadth-first search. Existing C++ language mechanisms and library components are expressive enough to essentially realize the algorithm line for line.

4.2 Requirements for Concrete Algorithms

A prototypical algorithm in this class is the breadth-first search (BFS) algorithm. The pseudocode for this algorithm, along with its C++ implementation, is shown in Figure 1. The algorithm is abbreviated from [8]. Modulo some type declarations that would be necessary for real code to compile, but which can be omitted from pseudocode, the C++ code, using out-of-the-box language mechanisms and library components, has essentially a one-one correspondence to the pseudocode.

From this implementation we can extract an initial set of requirements for the BFS algorithm:

- The graph `G` must meet the requirements of a *random access range*, meaning it can be indexed into with an object (of its difference type) and it has a size.
- The value type of `G` (the inner range of `G`) must meet the requirments of a *forward range*, meaning it is something that can be iterated over and have values extracted.
- The value type of the inner range must be something that can be used to index into `G`.
- All elements stored in `G` must be able to correctly index into it, meaning their value are between 0 and `size(G)-1`, inclusive.

Associated with the concepts of a random access range and forward range are complexity guarantees (which are also implied by the theoretical algorithm). Indexing into `G` is a constant-time operation and iterating over the elements in `G[u]` is linear in the number of elements stored in `G[u]`.

As an example, we could use any of the following compositions of standard library components for the `Graph` datatype above:

```
using Graph = std::vector<std::list<int>>;
using Graph = std::vector<std::vector<unsigned>>;
using Graph = std::vector<std::forward_list<size_t>>;
```

(In fact, any `Graph` data structure meeting the above requirements could be used.)

We now have a set of requirements for a concrete implementation of BFS. Following the generic programming process, there are various aspects of the implementation that we could begin lifting. Ultimately, as with the STL, we want a set of concepts useful across families

```
DIJKSTRA(G, w, s)
 1  for each vertex u ∈ V(G)
 2      d[u] ← ∞
 3      π[u] ← NIL
 4  d[s] ← 0
 5  Q ← V(G)
 6
 7  while Q ≠ ∅
 8      u ← EXTRACT-MIN(Q)
 9      for each v ∈ Adj(G)[u]
10          if d[u] + w(u, v) < d[v]
11              d[v] ← d[u] + w(u, v)
12              π[v] = u
```

```
void dijkstra(const Graph& G, int s) {...
  for (int u = 0; u < size(G); ++u) {
    d[u] = INF;
    pi[u] = NIL;                       }
  d[s] = 0
  for (int u = 0; u < size(G); ++u)
    Q.push({u, d[u]});
  while (!Q.empty()) {
    auto [u, x] = Q.top(); Q.pop();
    for (auto&& [v, w] : G[u]) {
      if (d[u] + w < d[v]) {
        d[v] = d[u] + w;
        pi[v] = u;                }}}}
```

Figure 2 Pseudocode and C++ implementation of Dijkstra's algorithm. As with BFS, existing C++ language mechanisms and library components are expressive enough to essentially realize the algorithm line for line.

of graph algorithms. So rather than lifting BFS in isolation, we now examine concrete implementations of other algorithms in order to identify common functionality that can be lifted in order to unify abstractions.

Figure 2 shows the pseudocode and corresponding C++ implementation for Dijksra's algorithm for solving the single-source shortest paths problem. From this implementation we can extract an initial set of requirements for the concrete `dijkstra` algorithm:

- The graph `G` is a *random access range*.
- The value type of `G` (the inner range of `G`) is a *forward range*.
- The value type of the inner range is a pair, consisting of a something we will call a vertex type and something we will call a weight.
- The vertex type is something that can be used to index into `G`.
- All values stored as vertex types in `G` must be able to correctly index into `G` meaning their value are between 0 and `size(G)-1`, inclusive.

Just as the code of `dijkstra` is similar to `bfs`, some of these requirements are also the same. **However, the key difference is in what is stored inside of the graph**. This implementation of `dijkstra` assumes that the graph stores a tuple consisting of *a vertex value and an edge weight*. That is, rather than the `Graph` types shown above, we could use the following for `dijkstra`:

```
using Graph = std::vector<std::list<std::tuple<size_t, int>>>;
using Graph = std::vector<std::vector<std::tuple<unsigned, double>>>;
using Graph = std::vector<std::forward_list<std::tuple<int, float>>>;
```

This is a different kind of graph than we had for `bfs`, which only stored a value. Yet, even a graph that stores a weight on its edge should be suitable for BFS exploration. Similarly, a graph without a weight on its edge should be suitable for Dijkstra's algorithm, provided a weight value can be provided in some way (or a default value, say, 1, used).

4.3 Lifting

From the foregoing discussion, we have two pieces of functionality we need to lift. First, we need to lift how the neighbor vertex is stored so that whether it is stored as a direct value or as part of a tuple (or any other way), it can be obtained. Second, we need to lift how

weights (or, more generally, **properties**) are stored on edges. And, finally, implied when we say we want to use different kinds of graphs with these algorithms, we need to parameterize them on the graph type (make them function templates rather than functions).

4.3.1 Parameterizing the Graph Type

In this lifting process we will be building up to a concept, which we will illustrate by lifting `dijkstra`. We begin by presenting its type parameterization. The prototype for a `dijkstra` function template based on our previous definition would be

```
template <class Graph>
auto dijkstra(const Graph& G, vertex_id_t<G> s);
```

Note that we have parameterized *two* things: the `Graph` type itself, as well as the type of the starting vertex `s`. In this case, the vertex type is not arbitrary, it is related to the type of the graph, and so we have a type primitive `vertex_id_t` that returns the type of the vertex associated with graph `G`.

We can update some of the previous requirements for the type-parameterized `dijkstra`:

- `Graph` must meet the requirements of `random_access_range`.
- The value type of `Graph` (the inner range of `Graph`) must meet the requirements of `forward_range`.
- The type `vertex_id_t<Graph>` is an associated type of `Graph`.
- The type `vertex_id_t<Graph>` is convertible to the `range_difference_t` of `Graph` (that is, it can be used to index into a `Graph`).

Both classes of graphs that we had previously seen for `bfs` and `dijkstra` satisfy these requirements (which are more general than either of the previous requirements). For example, both of the following compound structures

```
std::vector<std::vector<int>>;
std::vector<std::vector<std::tuple<int, float, double>>>;
```

satisfy the lifted requirements, (provided a suitable overload of `vertex_id_t` is defined) though each would have only satisfied one of the previous requirements. Note however, we still need to do more lifting before we can compose `bfs` or `dijkstra` with these types.

4.3.2 Lifting Neighbor Access

How a neighbor is stored is dependent on the graph structure itself; the mechanism for accessing it should therefore vary based on the graph type. In keeping with standard C++ practice – and since we want to be able to use C++ standard library containers, we adopt a polymorphic free function interface to abstract the process of accessing a neighbor. In particular, we define a `target` *customization point object* (CPO) to abstract how a neighbor vertex is accessed, given an object obtained from traversing the neighbor list.

- If variable `G` is of type `Graph` and variable `e` is of the value type of the inner range of `Graph`, then `target(G, e)` is a valid expression that returns a type of `vertex_id_t<Graph>`.
- All values returned by `target(G, e)` must be able to correctly index into a `Graph G`.

With this abstraction, the loop and neighbor access in `bfs` and `dijkstra` (respectively at lines 8 and 9 of Fig 2) are replaced by

```
for (auto&& e : G[u]) {
  auto v = target(G, e);
  ... }
```

Now, provided that suitable overloads for `target` are defined, the two model graph types

```
std::vector<std::vector<int>>;
std::vector<std::vector<std::tuple<int, float, double>>>;
```

will satisfy the above requirements, and we can compose them with the `bfs` and `dijkstra` we have lifted to this point. We can, for example, define overloads for target thusly:

```
int target(const std::vector<std::vector<int>>& G, int e) { return e; }
using E = std::tuple<int, float, double>;
int target(const std::vector<std::vector<E>>& G, E& e) { return std::get<0>(e); }
```

Note that these overloads are each specific to a single graph type. In practice we can define generalized overloads for entire classes of containers. In NWGraph we opted to realize `target` as a CPO, implemented using the `tag_invoke` mechanism [3].

4.3.3 Encapsulating Lifted Requirements as Concepts

We can encapsulate (and formalize) the above requirements in the form of a concept (which is almost a direct translation of the stated requirements to C++ code).

We first capture the very fundamental requirements of a graph, that it is a `semiregular` type (meaning that it is copyable and default-constructible) and that it has an associated `vertex_id_t` type:

```
template <typename G>
concept graph = std::semiregular<G>
    && requires(G g) { typename vertex_id_t<G>;  };
```

We define this as a separate concept since we may wish to define other concepts that reuse these requirements.

Next, we define some convenience type aliases to capture the type of the inner range of a graph as well as the type that is stored by the inner range:

```
template <typename G>
using inner_range = std::ranges::range_value_t<G>;
template <typename G>
using inner_value = std::ranges::range_value_t<inner_range<G>>;
```

Now we can define the concept that captures the requirements from the lifted `bfs` and lifted `dijkstra`:

```
template <typename G>
concept adjacency_list = graph<G>
    && std::ranges::random_access_range<G>
    && std::ranges::forward_range<inner_range_t<G>>
    && std::convertible_to<vertex_id_t<G>, std::ranges::range_difference_t<G>>
    && requires(G g, vertex_id_t<G> u, inner_value_t<G> e) {
  { g[u] } -> std::convertible_to<inner_range_t<G>>;
  { target(g, e) } -> std::convertible_to<vertex_id_t<G>>; };
```

Although we restricted our illustration of lifting to `bfs` and `dijkstra` in this paper, this concept captures the requirements for all algorithms in NWGraph based on adjacency lists (see also the discussion in Section 4.4).

We can use this concept to constrain the interface to `bfs` in the following two ways:

```
template <class Graph>
requires adjacency_list<Graph>
void bfs(const Graph& G);
```

```
template <adjacency_list Graph>
void bfs(const Graph& G);
```

When using concepts via the `requires` keyword, there is usually a fully general declaration and a number of abbreviated forms. In NWGraph, the second syntax above is preferred.

4.3.4 Lifting Edge Weight

In the concrete implementation of Dijkstra's algorithm shown above, we assumed the container associated with each vertex in the graph (i.e., the container obtained by `G[u]`) provided tuples containing the vertex id and the edge weight. In fact, there are numerous ways to associate a weight with each edge. We could, for example, store an edge index with each neighbor and use that to index into an array that we also pass into `dijkstra`. In such a case the (unconstrained) prototype for the algorithm might be

```
template <class Graph, class Range>
auto dijkstra(const Graph& G, vertex_id_t<Graph> s, Range wt);
```

The inner loop might then look like

```
for (auto&& e : G[u]) {
  auto v = target(e);
  auto w = wt[v];
  if (d[u] + w < d[v]) {
    d[v] = d[u] + w;
    pi[v] = u;          }}
```

To lift this version and the version with the directly-stored property on edges, we introduce `weights` as a parameter at the interface of `dijkstra`, of parameterized type `WeightFunction`. The `WeightFunction` template parameter is constrained by the `std::invocable` concept, which specifies that the function must be callable on an argument of the `inner_value` type of the `Graph`.

```
template <adjacency_list Graph, std::invocable<inner_value<Graph> WeightFunction>
auto dijkstra(const Graph& G, vertex_id_t<Graph> s, WeightFunction wt);
```

In this case, the inner loop would look like

```
for (auto&& e : G[u]) {
  auto v = target(e);
  auto w = wt(e);
  if (d[u] + w < d[v]) {
    d[v] = d[u] + w;
    pi[v] = u;          }}
```

4.3.5 About Vertex IDs

There is one aspect of the NWGraph `adjacency_list` concept that may seem overly restrictive, namely that the outer range of an `adjacency_list` be a random-access range and, hence, indexable by values in the range $[0, |V|)$. There are two reasons for this particular design decision. First, indexing into the outer range (`g[u]`) must be a constant-time operation in order for algorithms using `g[u]` to have their expected computational complexity (which is part of an algorithm's specification). Second, vertex ids are used not just for indexing into the graph itself, but for accessing vertex properties, which we also expect to be random-access ranges. This does not, however, necessarily imply that graph inner ranges must store vertex ids. Rather, the `adjacency_list` concept only requires that the `target` CPO return something convertible to a `vertex_id_t`, something that can be computed or looked up (though, again, in constant time). However, if one is going to compute a `vertex_id_t` on the fly, or look it up elsewhere, one could as well store it. NWGraph containers take this approach, and it can also be readily realized by nested standard library containers (e.g., `std::vector<std::vector<int>>`).

That all being said, the NWGraph `adjacency_list` constraints (like all concepts) are only syntactically enforced. Though unnecessary, as described above, one could provide a graph that used an `std::map` as the outer container. The operation `g[u]` would still work, but at the cost of increased computational complexity.

4.3.6 Non-Type Constraints

We have already seen in lifting the edge weight that not all constraints for an algorithm are encapsulated in the type requirements for the input graph. There are other requirements that an algorithm may have that cannot be captured as a type requirement, or as any compile-time checkable requirement. For example, some algorithms, such as triangle counting, may require that the edges within each neighborhood be sorted. Such requirements become part of the specification of the API, but cannot be made part of type checking. This is similar to, say, `binary_search` in the C++ standard library, which requires that the elements of the container to which it is applied be sorted. Yet, there is no such thing as a sorted container type in the standard library.

4.4 Other Graph Concepts

Our presentation thus far has developed a single concept (`adjacency_list`) and the reader may ask how broad that concept is, given the wide variety of potential graph algorithms. In fact, the `adjacency_list` concept is surprisingly broad in its applicability; only a few supplemental concepts are required to cover all of the algorithms implemented in NWGraph and probably all of the algorithms that are likely to be implemented in NWGraph in the future. This is perhaps not so surprising since the adjacency list $Adj(G)$ is also the primary theoretical construct upon which the majority of graph algorithms are built.

There are two additional concepts that we introduce briefly here which we found necessary for algorithms in NWGraph: `degree_enumerable` and `edge_list`. The former extends `adjacency_list` with the requirement that there be a valid expression `degree`, necessary in some algorithms. The latter is basically a container of objects for which `source` and `target` are valid expressions. Algorithms such as Bellman-Ford and Kruskal's MST use an edge list rather than an adjacency list [8].

Our confidence that these few concepts are sufficient is based on a comprehensive study of the concepts in the Boost Graph Library (BGL) [29]. The BGL has five essential graph concepts that cover all of its algorithms: `VertexListGraph`, `EdgeListGraph`, `AdjacencyGraph`, `IncidenceGraph`, and `BiIncidenceGraph`. Of these, the design decisions of NWGraph to require vertex identifiers to be indices obviates `VertexListGraph`; we don't need to iterate through a list of vertices provided by the graph, we simply iterate through vertex ids from 0 through $|V| - 1$. The NWgraph `adjacency_list` and `degree_enumerable` concepts subsume the essential functionality of `AdjacencyGraph` and `IncidenceGraph`. The `adjacency_list` does not have a `source` function requirement, but that is in fact only rarely used in the BGL algorithms requiring `IncidenceGraph` (and when it is used, there are other ways of obtaining the same information). The `BiIncidenceGraph` concept specifies that a graph type must have two lists of neighbors: those reachable by "out edges" (which is what `adjacejcy_list` requires) and those that can reach the vertex, i.e., the "in edges." The in edge neighborhoods are essentially the transpose of the out edge neighborhoods and can be represented with the same kind of adjacency list structure as the out edge neighborhoods. The need for a single data type holding lists of both out edges and in edges is unnecessary in the NWGraph design. Algorithms requiring a graph and its transpose take two graph arguments, one representing the out edges and one representing the in edges. Those two graphs represent (and store) exactly the same information as would be contained in a single `BiIncidenceGraph`, so there is no loss of efficiency in this design decision (and, in fact, NWGraph provides utilities for creating the transpose of a given graph). Finally, NWGraph includes an `edge_list` concept which is identical to the BGL `EdgeListGraph` concept.

5 Algorithms in NWGraph

Algorithms in NWGraph constitute the core of our library. NWGraph includes a broad classes of algorithms (sequential and parallel) for different graph problems, including graph traversal (BFS, SSSP), analytics (PageRank, Jaccard similarity, betweenness centrality, connected components), motif counting (triangle counting), network flow (maximum flow), etc. Table 1 lists the graph algorithms implemented in NWGraph along with their problem definitions.

5.1 Parallelization

NWGraph leverages existing parallelization support in the C++ standard library for implementing different parallel graph algorithms. However, in cases where it was necessary to circumvent some of the limitations of the C++ standard library for parallelization, we instead used Intel® oneAPI Threading Building Blocks (TBB) [14] for better performance.

5.1.1 Parallelization with `std` Execution Policies

NWGraph implements parallel algorithms for some of the different graph kernels described in Table 1 with `std::execution::par` (parallel policy) and `std::execution::par_unseq` (parallel unsequenced policy) provided to the `std::for_each` construct. Listing 5 demonstrates a triangle counting algorithm capable of benefiting from parallel `std::execution` policies. Note that updating shared variables relies on the `std::atomic` operations library.

Alternatively, Listing 6 shows an asynchronous task-based parallel triangle counting algorithm, which uses `std::future` and `std::async` to explicitly manage concurrency.

Table 1 Algorithm classes in NWGraph. Parallel implementation available: `std::execution` and `std::for_each`†, `std::async`§, TBB's `parallel_for`¶.

Algorithm	Definition
Breadth-first search†¶	Traverses a graph in breadth-fist search order from a given source. Implementation includes: top-down, bottom-up and direction-optimized [5] algorithms.
Depth-first search	Traverses a graph in depth-first search order from a given source.
Single-source shortest paths†¶	Finds the shortest distance paths from a given source to all other vertices in a graph. Δ-stepping algorithm [19] is implemented.
Connected component†¶	Finds connected components in a graph. Implementations include Afforest [32], Shiloach-Vishkin [26], BFS-based [27] and minimal label propagation [24, 34] algorithms.
PageRank†§¶	Compute the importance of each vertex in a graph. Implements the Gauss-Seidel algorithm [1].
Triangle counting†§¶	Counts the number of triangles in a graph. Implements algorithms discussed in [18].
Betweenness centrality†§¶	Measures how many times each vertex lies on the shortest paths to other vertices. Brandes Algorithm [7] has been implemented.
Maximum flow	Given a source and a sink, find paths with available capacity and push flow through them until there are no more paths available. Implements Edmonds-Karp algorithm.
K-core	Finds the subgraph induced by removing all vertices with degree less than k.
Jaccard similarity	Computes the Jaccard similarity coefficient of each pair of vertices in a graph.
Graph coloring	Assign a color to each vertex in the graph so that no two neighboring vertices have the same color. Implements Jones-Plassmann algorithm [15].
Maximal independent set	Graph coloring with two colors.

Listing 5 Parallel triangle counting algorithm with `std::execution` policies.

```
1  template <adjacency_list_graph Graph, class OuterExecutionPolicy =
2          std::execution::parallel_unsequenced_policy,
3          class InnerExecutionPolicy = std::execution::sequenced_policy>
4  std::size_t triangle_count(const Graph& A, OuterExecutionPolicy&& outer = {},
5          InnerExecutionPolicy inner = {}) {
6    std::atomic<std::size_t> total_triangles = 0;
7    std::for_each(outer, A.begin(), A.end(), [&](auto&& x) {
8      std::size_t triangles = 0;
9      for (auto &&i = x.begin(), e = x.end(); i != e; ++i) {
10       triangles += nw::graph::intersection_size(i,e,A[std::get<0>(*i)], inner);
11     }
12     total_triangles += triangles;
13   });
14   return total_triangles;
15 }
```

Listing 6 Parallel triangle counting algorithm with `std::async`.

```
template <class Op>
std::size_t triangle_count_async(std::size_t threads, Op&& op) {
  std::vector<std::future<size_t>> futures(threads);
  for (std::size_t tid = 0; tid < threads; ++tid) {
    futures[tid] = std::async(std::launch::async, op, tid);
  }
  // Reduce the outcome ...
}
template <adjacency_list_graph Graph>
std::size_t triangle_count_v2(const Graph& G, std::size_t threads = 1) {
  auto first = G.begin();
  auto last  = G.end();
  return triangle_count_async(threads, [&](std::size_t tid) {
    std::size_t triangles = 0;
    for (auto i = first + tid; i < last; i += threads) {
      for (auto j = (*i).begin(), end = (*i).end(); j != end; ++j) {
      //  ...
}} });}

```

5.1.2 Shortcomings of `std` Execution Policy-based Parallelization

The current `std::execution` and `std::thread` libraries lack adequate support for implementing efficient parallel graph algorithms. Some of the most important limitations include:

- Programmers do not have control over workload distribution or partitioning of data or work among threads.
- Thread-safe data structures are not part of the standard library. Having to manually use coarse-grained locking `lock` and `mutex` to make standard library containers thread-safe is labor-intensive and may severely limit the performance of parallel graph algorithms.
- Granularity of concurrency cannot be directly managed.

5.1.3 Parallelization with Intel® Threading Building Blocks

To circumvent these shortcomings, NWGraph leverages Intel® Threading Building Blocks (TBB) library. TBB provides a set of efficient concurrent containers (`hashmap`, `vector`, and `queue`) implemented with fine-grained locking and lock-free techniques. NWGraph uses TBB's concurrent vector to maintain the frontier list of active vertices in each step of the Δ-stepping algorithm [19] for computing SSSP (Listing 7).

One determinant of parallel graph algorithm performance is how well the parallel workload is balanced among threads. Graph algorithms typically do not perform well with naive partitioning approaches. Recall a graph structure is a random-access range of forward ranges. A naive partitioning scheme will partition the outer range into equal-sized chunks – which is a reasonable strategy for one-dimensional containers, where each partition will have essentially the same amount of work. The story is completely different for graph data structures, especially those with highly skewed degree distributions, such as power-law graphs. In such cases, if the graph is partitioned based on the outer range, each partition will have the same number of starting vertices (the same number of inner ranges), but the number of neighbors

Listing 7 Δ-stepping algorithm for computing single-source shortest paths using TBB's thread-safe containers.

```
1  template <class distance_t, adjacency_list_graph Graph, class Id, class T>
2  auto delta_stepping(const Graph& graph, Id source, T delta) {
3    tbb::queuing_mutex                                 lock;
4    tbb::concurrent_vector<tbb::concurrent_vector<Id>> bins(size);
5    tbb::concurrent_vector<Id> frontier;
6    // ...
7    while (top_bin < bins.size()) {
8      frontier.resize(0);
9      std::swap(frontier, bins[top_bin]);
10     tbb::parallel_for_each(frontier, [&](auto&& u) {
11       if (tdist[u] >= delta * top_bin) {
12         nw::graph::parallel_for(graph[u], [&](auto&& v, auto&& wt) {
13             relax(u, v, wt); });
14       } });
15     // ...
16   }
17 }
```

Listing 8 Δ-stepping algorithm for computing single-source shortest paths using TBB's `blocked_range` partitioning technique.

```
1  // ...
2  while (top_bin < bins.size()) {
3    // ...
4    tbb::parallel_for(tbb::blocked_range(0ul,frontier.size()),[&](auto&& range){
5      for (auto id = range.begin(), e = range.end(); id < e; ++id) {
6        auto i = frontier[id];
7        if (tdist[i] >= delta * top_bin) {
8          // ...
9    }}});
10 }
```

in each inner range will vary with the degree distribution. Without the ability to partition based on the size of the inner ranges (which is an indication of the amount of work to be done for each partition), some threads may end up with vastly more work than other threads.

To provide better control of workload distribution among threads, TBB's `parallel_for` function accepts ranges (a TBB construct in this case, not to be confused with C++20 ranges) that can be customized to provide user-defined partitioning. An example is NWGraph's use of TBB's `blocked_range` in the Δ-stepping algorithm (compare Listing 7 with Listing 8). Custom ranges are not limited to contiguous partitions of the underlying data structure. Instead, one can use strided (or cyclic) partitions – or, more generally, block-cyclic partitions – which can provide natural load balancing in certain situations. We show the performance benefit of cyclic distribution in Section 8.

TBB also implements C++ standard library parallelism (TBB's parallel STL). Intel has open-sourced TBB (now called oneTBB), allowing its parallel STL effecting parallelism in other C++ library implementations. In fact, and somewhat ironically, the standard library provided by g++ (the compiler we used for NWGraph development) is one such compiler that uses TBB under the hood.

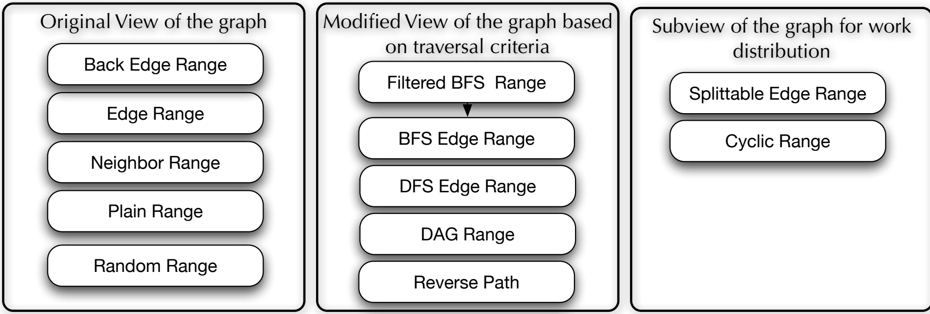


Figure 3 Range adaptors in NWGraph.

6 Graph Range Adaptors in NWGraph

A key feature of the new C++ Ranges is the notion of *views*, which allow for different ways to access data in a range without changing the underlying range. Between a range and a range view sits a *range adaptor*, which takes the original range and presents it to the user as a view while hiding the underlying data manipulation details. We leverage range adaptors to simplify graph algorithms in NWGraph, by providing reusable data access patterns that eliminate the need for visitor objects.

Consider again BFS traversal, a core graph algorithm kernel. Except perhaps for benchmarking, a standalone BFS traversal is rarely useful. Rather, other algorithms use a BFS traversal pattern to perform more useful computations, such as finding the distance to every vertex from the source, finding the parent list, etc. One approach to applying BFS traversals to other types of computations would be to further parameterize `bfs` with additional functions. However, what to apply and where to apply it is not well defined – we don't necessarily have well-defined concrete algorithms to lift.

The Boost Graph Library provides extensibility to BFS through its *Visitor* mechanism, which is essentially a large structure with callbacks used at multiple entry points in BFS execution [29]. The BFS Visitor has nine different possible callbacks, making actual extension of the BGL BFS a complicated proposition.

NWGraph does not attempt to further lift algorithms from arbitrary, concrete use cases (which are not well-defined from a library designer's perspective). Instead it provides range adaptors that allow the graph to be iterated over in a specified order (either vertex by vertex or edge by edge). For example, NWGraph provides `bfs_range` for traversing the vertices of a graph in breadth-first order, and `bfs_edge_range` for traversing the edges.

```
for (auto&& u : bfs_range(G)) { /* visit vertex u */ }
for (auto&& [u, v] : bfs_edge_range(G)) { /* Visit edge u,v */ }
```

As views are concise and efficient ways of representing the same data in multiple ways, graph algorithms can be considered as operating on a range of elements of a graph with different requirements on how data is being viewed by the algorithm. In NWGraph, we provide three categories of view of the graph shown in Figure 3:

- *Original view of the graph:* These include edge range, neighbor range, plain range, random range and back-edge range.
- *Modified view of the graph based on traversal criteria:* For example, BFS and DFS traversal-based algorithms consider vertices in a certain order. These alternative views include BFS edge range, filtered BFS range, DFS edge range, Directed Acyclic Graph

(DAG) range and Reverse Path. More sophisticated range adaptors such as DAG range, for example, iterate over the vertices in a particular order, based on the predecessor-successor relationships, imposed by algorithm-specific heuristics.
- *Subview of the graph for workload distribution in parallel execution:* These include splittable edge range and cyclic range.

7 Model Data Structures in NWGraph

In Section 4.3 (lifting), we demonstrated that the built-in types in the standard library are sufficient to construct a graph. We reiterate that any data structure meeting the requirements specified by the NWGraph concepts can be composed with the NWGraph algorithms based on those concepts. For instance, `std::vector<std::vector<std::tuple<size_t, double>>>` meets the requirements of the `adjacency_list` concept, and hence can be used with any of the appropriate algorithms. Graphs do not need to be constructed *from* a range of ranges in order to meet the requirements *of* a range of ranges. Data structures such as compressed sparse structures, which represent all of a graph's neighborhoods contiguously in memory, can offer better performance due to more favorable memory accesses. We compare compressed sparse structures to compositions of standard library components in Section 8.

The workhorse graph structure for NWGraph is the class template `nwgraph::adjacency`, a compressed structure with the following (abbreviated) interface:

```
template <int idx, class Attributes...>
  class adjacency {
    class outer_iterator {
      using iterator_category =
      std::random_access_iterator_tag; ..};
    class inner_iterator;
    outer_iterator begin();
    outer_iterator end();
    operator[](index_t i) const;  };
```

`nwgraph::adjacency` is parameterized on the types of the edge properties, using variadic template parameter `Attributes`, to allow an arbitrary number of edge properties of arbitrary type. The `idx` parameter is a hint indicating whether the adjacency structure is representing the out edges or the in edges of the edge list from which it was built. To allow `nwgraph::adjacency` to meet the requirements of `adjacency_list` (range of ranges), we define a private iterator type that acts as a random access iterator.

NWGraph has a small set of utility functions for building graphs from a given dataset in a generic fashion. The first step involves building an index edge list, given a vertex table and an edge table. The second step uses this edge list to build an index graph (that is, filling in a structure modeling adjacency). Some algorithms (such as triangle counting) require sorted data in the neighborhood range. The graph construction algorithms take a runtime flag that indicates whether neighborhood sorting should be done during graph construction. NWGraph also provides functions to sort graphs that have already been constructed. The pertinent APIs for graph construction are the following:

```
template <class IndexEdgeList, class VRange, class ERange>
IndexEdgeList make_index_edge_list(const VRange& vertices, const ERange& edges);
template <adjacency_list Graph, class IndexEdgeList>
Graph make_graph(const IndexEdgeList& edge_list);
```

8 Performance Evaluation

8.1 Experimental Setup

Our experiments were carried out on compute nodes consisting of two Intel® Xeon® Gold 6230 processors, each with 20 physical cores running at 2.1 GHz (with turbo boost up to 3.9GHz), and hyperthreading disabled. Each processor has 28MB L3 cache and 188GB of main memory. NWGraph is implemented in C++20, parallelized with oneTBB 2021.4, and compiled with the g++ 11.2 compiler using `-Ofast -march=native` compilation flags.

8.2 Abstraction Penalty

Modern C++ practice includes a wide variety of mechanisms and related idioms for traversing data structures. Since the inner range of a type meeting `adjacency_list` requirements is a forward range, any of those modern techniques may be used for traversal. Moreover, the compressed graph structure provided in NWGraph presents a facade of being a range of ranges, using internally-provided iterators to effect the "range of ranges" interface. Given this variety of traversal mechanisms, and the layers of abstraction associated with traversal and with the compressed graph structure, there is potential for unintended abstraction penalty.

To verify the performance expectation of specialization in generic libraries, i.e., that there is minimal abstraction penalty, NWGraph includes an abstraction penalty benchmark suite, from which we present a small subset. Here, we focus on inner range traversal as it is ubiquitous to all graph algorithms; any penalties uncovered there would also be apparent in other graph algorithms. We use the sparse matrix-vector product (SpMV) algorithm as the vehicle for our study, as it is well-suited for characterizing inner range traversal; it makes one pass through the entire graph, traversing each of the inner ranges.

Let us consider a "raw for loop" implementation of SpMV, using a compressed sparse row (CSR) data structure to store the adjacency list. The CSR structure stores its neighbor indices and edge weights in contiguous arrays and traverses the data structure by looping through each vertex id and then traversing the associated inner range delimited by the indices in the `ptr` array.

```
auto ptr = G.indices_.data();
auto idx = std::get<0>(G.to_be_indexed_).data();
auto dat = std::get<1>(G.to_be_indexed_).data();
for (vertex_id_t i = 0; i < N; ++i) {
  for (auto j = ptr[i]; j < ptr[i + 1]; ++j) {
    y[i] += x[idx[j]] * dat[j]; }}
```

This concrete algorithm establishes the baseline performance against which the generic algorithms are compared.

In a generic SpMV implementation, we cannot assume this underlying CSR structure. Rather we can only assume the interface specified by the `adjacency_list` concept, i.e., a range of ranges, and our implementations of a generic SpMV must be written accordingly. However, to meet our specialization performance requirements, a generic SpMV written to the `adjacency_list` concept must still provide the same performance as the concrete baseline when composed with a CSR-like structure, i.e., the NWGraph compressed graph `adjacency` structure.

Consider two common iteration patterns used in modern C++, an iterator-based for loop and a range-based for loop (which is essentially syntactic sugar for the iterator-based loop):

```
vertex_id_t k = 0;
for (auto i = G.begin(); i != G.end(); ++i) {
  for (auto j = (*i).begin(); j != (*i).end(); ++j) {
    y[k] += x[get<0>(*j)] * get<1>(*j); }
  ++k; }
```

```
vertex_id_t k = 0;
for (auto&& i : G) {
  for (auto&& [j, v] : i) {
    y[k] += x[j] * v; }
  ++k; }
```

As generic loops, these can be applied to any graph that models the `adjacency_list` concept. There are several important departures from the concrete CSR-based loops. It is easy to see how these operate on something that is a range of ranges. On the other hand, there is no obvious correspondence between the iterator-based algorithms and the concrete algorithm. Of particular note is that the neighbor vertex index `j` and the edge weight `v` are accessed as tuples, directly in the former case and via structured binding in the second case.

Iterators can also be used to traverse the inner range using the standard library `std::for_each` algorithm rather than for loops. The `std::for_each` algorithm iterates through the indicated iterator range and applies a given function to each element in the range. Here, we specify those functions using C++ lambdas.

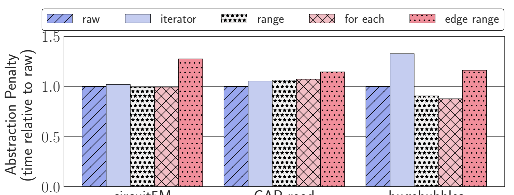
```
vertex_id_type k = 0;
std::for_each(graph.begin(), graph.end(), [&](auto&& nbhd) {
  std::for_each(nbhd.begin(), nbhd.end(), [&] (auto&& elt) {
    auto&& [j, v] = elt;
    y[k] += x[j] * v; });
  ++k; });
```

In the previous examples, we iterate through the graph using two nested loops, variously expressed. We can alternatively use the `edge_range` range adaptor, which "flattens" the graph, allowing traversal of all of the inner ranges with a single loop.

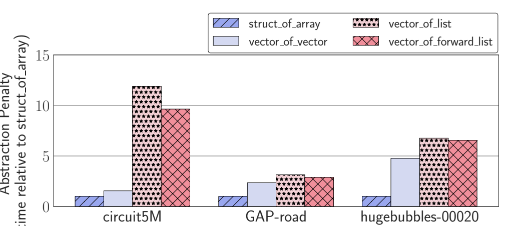
```
for (auto [i, j, v] : make_edge_range<0>(graph))
  y[i] += x[j] * v;
```

The `edge_range` adaptor essentially turns the `adjacency_list` into an `edge_list`. It provides a tuple with three elements: The source vertex, the target vertex, and the edge weight. The result is an extremely concise implementation of SpMV, which, again, will work with any type meeting the requirements of `adjacency_list`. The question that we wish to address is whether this genericity and this conciseness comes at the cost of performance.

Our experimental evaluation of SpMV uses three graphs with different underlying topology taken from the SuiteSparse matrix collection: `circuit5M`, `GAP-road`, and `hugebubbles` [9]. These graphs have similar numbers of edges (30M to 60M) and the benchmarks run in comparable time. Figure 4a shows the results of the different data access abstractions relative to the raw loop timing, for each benchmark. Timing results were averaged over 5 runs of each benchmark. Bars significantly higher than the raw for loop bar would indicate a significant performance penalty. None of the abstraction methods incurs a significant performance



(a) Different data access abstractions (`iterators`, `ranges`, `std::for_each` and `edge_range` adaptor) with their abstraction penalties measured relative to a raw for loop implementation. There is no significant performance penalty relative to the raw loop implementation.



(b) Measured abstraction penalty for the SpMV benchmark with graphs represented by different containers using `iterator` based `for` loop. The execution time has been normalized w.r.t to the execution time of SpMV with graphs represented as `struct_of_array` (lower is better).

Figure 4 Abstraction Penalty Benchmarks with SpMV.

Table 2 Characteristics of input graphs used for performance evaluation.

Name	Description	#Vertices (M)	#Edges (M)	Degree Distribution	References
`road`	USA road network	23.9	57.7	bounded	[11]
`twitter`	Twitter follower Links	61.6	1,468.4	power	[17]
`web`	Web Crawl of .sk Domain	50.6	1,930.3	power	[6]
`kron`	Synthetic Graph	134.2	2,111.6	power	[20]
`urand`	Uniform Random Graph	134.2	2,147.5	normal	[12]

penalty relative to the raw loop implementation. `edge_range` is perhaps consistenly a little higher than the baseline, due to moving access of the row index from the outer loop to the inner loop. Continued refinement of `edge_range` is a topic of ongoing work.

8.3 Graph Representations

We also evaluated the performance implications of different choices for the inner range: `adjacency`, `vector_of_vector`, `vector_of_list`, and `vector_of_forward_list`. The latter three graph structures are lightweight wrappers around the corresponding composed standard library containers, and provide a variadic interface to match `adjacency`. Note that all of these containers meet the requirement of our `graph` concept. However, they have different features outside of the context of graph algorithms that might make them suitable for different situations. Notably they can represent more dynamic graphs, i.e., they can be modified (vertices or edges added or deleted) much more efficiently than the compressed form.

This flexibility comes at a cost. Figure 4b shows the performance of the iterator-based SpMV on the different containers. Execution time is normalized relative to SpMV with the `adjacency` container. Unlike the results in Figure 4a, there are significant differences in performance between the different cases. Note, however, that these experiments are not measuring the difference between an abstract and a concrete expression of an algorithm. Rather, the generic algorithm is the same in each of the cases, but it is composed with different data structures. The benchmark compares the time it takes to traverse the different inner range structures (vector, doubly-linked list, singly linked list). The `adjacency` representation is cache-friendly, supporting efficient access of the outer and inner range, while the performance of the other graph types reflect the expected overheads of their underlying inner ranges.

Table 3 GAP Benchmark Suite execution times for NWGraph.

Algorithm	kron	urand	twitter	web	road
BFS	0.51s	1.12s	0.26s	0.72s	0.84s
SSSP	7.23s	13.20s	2.88s	2.02s	2.99s
CC	0.64s	1.50s	0.29s	0.32s	0.09s
PR	12.09s	12.45s	8.69s	2.81s	0.26s
BC	8.43s	12.70s	2.42s	1.31s	1.95s
TC	305.19s	20.42s	58.81s	7.40s	0.07s

Table 4 Performance comparisons with NWGraph for the GAP Benchmark Suite. Percentages represent the relative speedup of each particular experiment relative to the NWGraph. The color code indicates performance that is lower than (red), equal to (white), or higher than (green) NWGraph.

	Galois					GraphIt					GAPBS				
	kron	urand	twitter	web	road	kron	urand	twitter	web	road	kron	urand	twitter	web	road
BFS	118%	168%	40%	242%	886%	9.2%	9.5%	1.8%	8.2%	57%	174%	203%	168%	240%	218%
SSSP	136%	131%	108%	147%	1162%	100%	202%	114%	DNR	2.3%	189%	228%	250%	261%	1339%
CC	104%	90%	101%	174%	154%	0.34%	135%	0.53%	75%	0.02%	151%	112%	164%	174%	10%
PR	86%	65%	78%	99%	73%	4.5%	4.0%	4.6%	5.8%	4.8%	137%	116%	828%	101%	9.1%
BC	OOM	28%	0.08%	1.4%	44%	OOM	OOM	OOM	OOM	OOM	37%	38%	45%	40%	110%
TC	80%	71%	84%	56%	240%	107%	OOM	86%	OOM	120%	90%	98%	130%	54%	42%

8.4 Performance on Large-Scale Graphs

To demonstrate NWGraph's performance characteristics on large-scale graphs, we evaluate and compare NWGraph with three well-established high-performance graph frameworks: GAP [4], Galois [22] and GraphIt [36], on the algorithms and graphs that comprise the GAP benchmark suite [4]. The algorithms in the benchmark are betweenness centrality (BC), breadth-first search (BFS), connected components (CC), PageRank (PR), single source shortest path (SSSP), and triangle counting (TC). The graphs used in the benchmark (shown in Table 2) are large, with diverse structural properties. All experiments were conducted with 32 threads running on 32 physical cores. The frameworks have been previously tuned for the GAP benchmark suite and were run under carefully controlled conditions, according to the rules and procedures established in [2].

Speedups of the different graph frameworks over NWGraph for the five datasets is shown in Table 4. We summarize our observations as follows:

- NWGraph outperforms the other frameworks in the majority of cases for BC and TC. The TC implementation has been highly tuned, using a cyclic range adaptor for effective load balancing, as well as having efficient implementations of its pre-processing techniques (which time is included in the benchmarking), such as relabeling the vertices by degree [18].
- NWGraph is better than Galois and GraphIt for PR, and somewhat worse than GAPBS. NWGraph and GAPBS both implement PR using a more efficient Gauss-Seidel inner step in the algorithm.
- For BFS and SSSP, NWGraph does not perform as well as Galois or GAPBS, particularly for `road`, for which Galois's highly-asynchronous approach is particularly effective. We do not currently have an explanation for NWGraph's poor performance on `road`.
- All frameworks except GraphIt implement the Afforest algorithm [32] for CC. Hence, GraphIt's CC performs poorly for graph inputs having large dominant components.

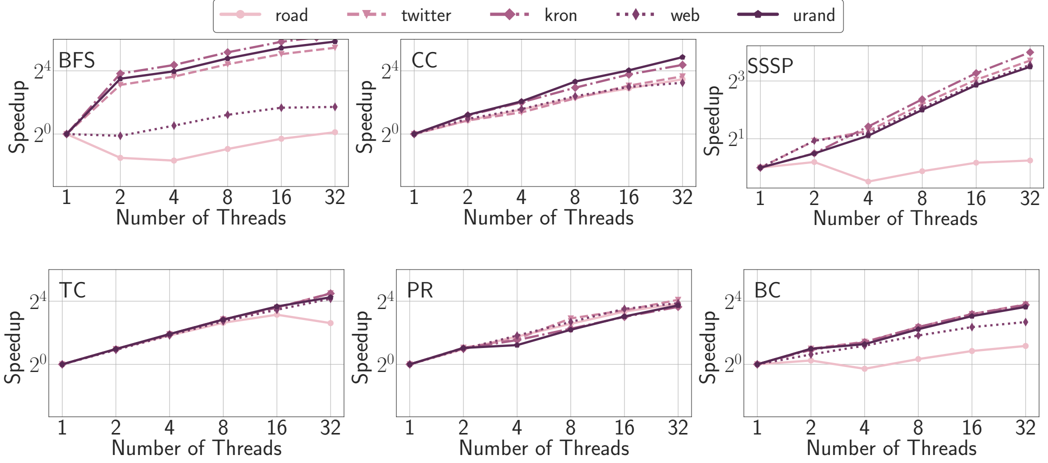


Figure 5 Strong scaling performance of six different graph algorithms (BFS, CC, TC, PR, PR, and BC) with five GAP graph inputs. The reported speedup is calculated as the ratio of the sequential (single-threaded) execution time and the parallel execution time.

One takeaway from these results is that the choice of algorithm and how well it is matched to a particular graph have the largest effect on performance. The performance differences between NWGraph and other frameworks (better or worse) are not due to inherent properties of the C++ language, nor its standard library, upon which NWGraph is built.

8.5 Strong Scaling Performance

Figure 5 presents the strong scaling performance for the graph kernels and inputs from the GAP benchmark [4]. For strong scaling, we keep the (size of the) dataset fixed while increasing the number of threads. The reported speedup is calculated by taking the ratio of the sequential execution time and the parallel execution time. In most cases, the algorithms scale well. Two exceptions can be observed with the road network input for the BFS and the SSSP algorithms. Since the road network has a low average degree and large diameter, increasing the number of threads does not improve the performance of these two algorithms significantly. BFS with web graph also does not demonstrate expected scalability.

8.6 Comparison with Boost Graph Library

We have compared NWGraph to BGL several times in this paper with respect to certain design decisions. Of interest also is how NWGraph performance would compare to BGL. We compare the sequential performance of the two libraries for four of the GAP graph kernels using the GAP graph inputs in Table 2. We report the results in Table 5. As can be observed from the Table, NWGraph performs better than BGL in all cases. (BGL has no directly comparable implementations of BC mor PR, and hence we are unable to compare the performance for these two kernels.)

9 Related Libraries and Toolkits

This section explores the landscape of related graph libraries and frameworks. Each of the libraries or tools discussed in this section make different design tradeoffs regarding usability, extensibility, and performance. Though few of the tools in this section (with the exception of BGL) aimed to fill the role of an STL graph library, they all contribute to a greater understanding of graph library design.

Table 5 Sequential runtime and speedup of NWGraph and BGL for four graph algorithms: TC, CC, BFS, and SSSP. >24H indicates jobs that did not finish within 24 hours; OOM indicates out of memory.

Algorithm	Library	road	twitter	kron	web	urand
TC	BGL	1.34s	>24H	>24H	>24H	4425.54s
	NWGraph	0.41s	1327.63s	6840.38s	131.47s	387.53s
	Speedup	3.27	-	-	-	11.42
CC	BGL	1.36s	21.96s	81.18s	6.64	134.23
	NWGraph	1.02s	3.65s	13.37s	3.02s	43.74s
	Speedup	1.34	6.02	6.07	2.20	3.07
BFS	BGL	1.09s	12.11s	54.80s	5.52s	73.26s
	NWGraph	0.91s	11.25s	38.86s	2.37s	64.63s
	Speedup	1.20	1.08	1.41	2.33	1.13
SSSP	BGL	4.03s	47.89s	167.20s	28.29s	OOM
	NWGraph	3.35s	40.94s	95.06s	23.51s	177.13s
	Speedup	1.21	1.17	1.76	1.20	-

Generic C++ Graph Libraries. BGL [29] and the LEMON graph library [10] both contributed to the development of generic graph algorithms in C++. BGL proposed algorithm templates that could be used on a variety of graph types (which could be generated using BGL's graph type generator), e.g., vector of lists, list of vectors, etc. Vertices and edges were allowed to be arbitrary types accessed via property maps which could be stored internally or externally to the graph. The default graph algorithms could be customized using visitor objects, which allowed users to use existing data access patterns to do additional work. LEMON shared many of these features. Both libraries advertise algorithms that work with user-defined graphs, so long as they conform to a certain interface.

Some of these features had shortcomings that limited their use. The visitor objects are difficult to use, both from a programming and algorithmic design perspective. Property maps are a powerful programming abstraction, but in addition to being difficult to use, could lead to performance issues. The type of LEMON's graph adaptors are different from the original graph type being adapted, and their use as graphs is only supported in limited ways. A major shortcoming of these designs is the difficulty of using custom data structures. In order to adapt an existing user-defined data structure, the BGL interface requires overloading several global free functions. These mostly include accessors, mutators, and iterators for edges and vertices. An assumption is placed on the graph container type being adapted that it will have much of the same behavior as the built in BGL container types. Furthermore both libraries lack newer features in C++ such as `constexpr`, variadic templates, automatic type deduction, execution policies, etc.

HPC Graph Frameworks. There are several graph frameworks designed to maximize performance in distributed memory or shared memory, such graph frameworks include Parallel Boost Graph Library (PBGL) [13], Galois [16], Ligra [28], Giraph [25], Gunrock [33], GraphIt [35], etc. The contributions of these frameworks are typically a computational model for parallel processing of graphs, including on clusters and GPUs, with less emphasis on the usability or extensibility of graph algorithms or containers. A thorough evaluation of several well-known parallel graph frameworks can be found in [2].

10 Conclusion

In this paper we presented the design and rationale for a modern generic C++ library of graph algorithms and data structures, NWGraph. Based on a careful analysis of the graph problem domain, the fundamental interface abstraction underlying NWGraph is that of a random access range of forward ranges. Intentionally minimal, this interface admits composition with any types that meet its requirements. The library implementation includes selected concreted containers and a rich selection of common graph algorithms. Though the library is implemented with standard library components using idiomatic C++, experimental results showed that the interfaces present no abstraction penalty and that the NWGraph implementation has performance on par with the highest performing competition. We intend to continue to refine NWGraph and use it as a testbed in support of an emerging proposal to the C++ standards committee for a standard C++ graph library.

References

1. Arvind Arasu, Jasmine Novak, Andrew Tomkins, and John Tomlin. PageRank computation and the structure of the web: Experiments and algorithms. In *WWW*, pages 107–117, 2002.
2. Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Mark Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, Tim Davis, Kevin Deweese, Jesun Firoz, et al. Evaluation of graph analytics frameworks using the gap benchmark suite. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 216–227. IEEE, 2020. doi:10.1109/IISWC50251.2020.00029.
3. Lewis Baker, Eric Niebler, and Kirk Shoop. tag_invoke: A general pattern for supporting customisable functions. Technical Report P1895R0, JTC1, 2019. URL: http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1895r0.pdf.
4. Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. *arXiv*, 2015. doi:10.48550/ARXIV.1508.03619.
5. Scott Beamer, Krste Asanović, and David A. Patterson. Direction-optimizing breadth-first search. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, 2012. doi:10.1109/SC.2012.50.
6. Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. *WWW*, pages 595–601, 2004. doi:10.1145/988672.988752.
7. Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001. doi:10.1080/0022250X.2001.9990249.
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 3rd ed edition, 2009.
9. Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. doi:10.1145/2049662.2049663.
10. Balázs Dezső, Alpár Jüttner, and Péter Kovács. Lemon–an open source c++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45, 2011. URL: https://lemon.cs.elte.hu/trac/lemon.
11. 9th DIMACS implementation challenge - Shortest paths, 2006. URL: http://www.dis.uniroma1.it/challenge9/.
12. Paul Erdős and Alfréd Rényi. On random graphs. I. *Publicationes Mathematicae*, 6:290–297, 1959.
13. Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 423–437, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094844.
14. Intel. Intel Threading Building Blocks (TBB), 2020. URL: https://github.com/oneapi-src/oneTBB.

15 Mark T Jones and Paul E Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993. doi:10.1137/0914041.

16 Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222. ACM, 2007. doi:10.1145/1250734.1250759.

17 Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM. doi:10.1145/1772690.1772751.

18 Andrew Lumsdaine, Luke Dalessandro, Kevin Deweese, Jesun Firoz, and Scott McMillan. Triangle counting with cyclic distributions. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2020. doi:10.1109/HPEC43674.2020.9286220.

19 Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. 1998 European Symposium on Algorithms. doi:10.1016/S0196-6774(03)00076-2.

20 Richard C. Murphy, Kyle B. Wheeler, Brian W Barrett, and James A. Ang. Introducing the Graph 500. In *Cray User's Group*. CUG, 2010.

21 David R. Musser and Alexander A. Stepanov. Generic programming. In P Gianni, editor, *International Symposium ISSAC 1988*, volume 38 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1989.

22 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM. doi:10.1145/2517349.2522739.

23 Eric Niebler, Casey Carter, and Christopher Di Bella. The one ranges proposal. Technical report, Tech. rep. P0896r4. Nov. 2018., 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4.pdf.

24 S. M. Orzan. *On Distributed Verification and Verified Distribution*. Ph.d. thesis, VRIJE UNIVERSITEIT, November 2004. URL: http://dare.ubvu.vu.nl/handle/1871/10338.

25 Roman Shaposhnik, Claudio Martella, and Dionysios Logothetis. *Practical Graph Analytics with Apache Giraph*. Apress, New York, 1st ed. edition edition, October 2015.

26 Yossi Shiloach and Uzi Vishkin. An O(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982. doi:10.1016/0196-6774(82)90008-6.

27 J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 143–153. ACM, 2014. doi:10.1145/2612669.2612692.

28 Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2442516.2442530.

29 Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

30 Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-95-11, HP Laboratories, November 1995. URL: http://stepanovpapers.com/STL/DOC.PDF.

31 Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

32 Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21. IEEE, 2018. doi:10.1109/IPDPS.2018.00012.

33 Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. ACM. doi:10.1145/2851141.2851145.

34 Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proc. VLDB Endow.*, 7(14):1821–1832, 2014. doi:10.14778/2733085.2733089.

35 Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with GraphIt. In *CGO*, pages 158–170. ACM, 2020. doi:10.1145/3368826.3377909.

36 Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276491.

Vincent: Green Hot Methods in the JVM

Kenan Liu [1] ✉
SUNY Binghamton, NY, USA

Khaled Mahmoud [1] ✉
SUNY Binghamton, NY, USA

Joonhwan Yoo ✉
SUNY Binghamton, NY, USA

Yu David Liu ✉
SUNY Binghamton, NY, USA

Abstract

In this paper, we show the energy efficiency of Java applications can be improved by applying Dynamic Voltage and Frequency Scaling (DVFS) inside the Java Virtual Machine (JVM). We augment the JVM to record the energy consumption of *hot methods* as the underlying CPU is run at different clock frequencies; after all the frequency possibilities for a method have been explored, the execution of the method in an optimized run is set to the CPU frequency that leads to the most energy-efficient execution for that method. We introduce a new sampling methodology to overcome the dual challenges in our design: both the underlying measurement mechanism for energy profiling and the DVFS for energy optimization are overhead-prone. We extend JikesRVM with our approach and benchmark it over the DaCapo suite on a server-class Linux machine. Experiments show we are able to use 14.9% less energy than built-in power management in Linux, and improve energy efficiency by 21.1% w.r.t. the metric of Energy-Delay Product (EDP).

2012 ACM Subject Classification Software and its engineering → Software performance

Keywords and phrases energy efficiency, JVM, just-in-time compilation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.32

Category Extended Abstract

Related Version *Full Version*: `http://www.cs.binghamton.edu/~davidl/papers/ECOOP22Long.pdf`

Supplementary Material *Software (Source Code and Data)*: `https://bitbucket.org/vincent-paper/vinccent/`
archived at `swh:1:dir:1cb44124eadec1ce51dca40c323bd388da57ea0c`

Funding This project is supported by the US NSF award CNS-1910532.

1 Introduction

The carbon footprint of data centers has recently received significant scrutiny [42]. After mobile workloads, server-class workloads once again place energy-efficient computing in the spotlight. This design goal is addressed at many layers of the computing stack. Among them, a less explored approach is to study the energy impact of *managed runtimes*, a middle layer between high-level applications and low-level systems. Relative to lower-layer techniques on hardware design (e.g., [17]) and OS design (e.g., [60]), a runtime approach has the benefit of guiding energy optimization with runtime-specific information. Relative to higher-layer techniques e.g., energy-aware programming languages [55, 10, 49, 26, 19, 11, 34, 25, 41, 61, 15],

[1] These authors are currently affiliated with Intel Corporation.

© Kenan Liu, Khaled Mahmoud, Joonhwan Yoo, and Yu David Liu;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 32; pp. 32:1–32:30
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a runtime approach can work with programs written in existing languages, arguably easier for adoption. In a nutshell, the runtime – strategically positioned between the lower layers and the higher layers – can often combine the benefits of both sides of its neighbors on the computing stack.

At their essence, all runtime-based approaches are motivated by the same question: what information uniquely available in the runtime can be harvested to guide energy optimization? As examples, existing efforts have relied on thread and synchronization states (e.g., [2]), just-in-time (JIT) compilation strategies (e.g., [56]), and garbage collector (GC) designs (e.g., [29]) to inform energy optimization.

1.1 Our Approach: JVM-Level Method-Grained DVFS

We introduce a novel energy optimization at the level of the JVM. It relies on two basic facts of the JVM: (i) the JVM is aware of the boundary of programming abstractions such as methods; (ii) the JVM is aware of how often a method is used. Both pieces of information are readily available among existing JVMs, good news for the adoption of our approach.

Our key idea is *method-grained energy optimization*: it demarcates the boundary of DVFS [27, 13] adjustment with the boundary of methods. Our premise is that each method as a logical unit of the program behavior can serve as an ideal granularity for energy optimization. For example, the method `Matrix4.transformP` in a ray-tracing benchmark `sunflow` [12] may carve out the boundary of a CPU-intensive computation, and the method `PSStream.write` in a file processing benchmark `fop` [12] may demarcate an I/O-intensive computation. It is well known that energy optimization based on DVFS can be effectively performed based on program *phased behaviors* [52, 53, 32], i.e., an application may go through phases of different levels of CPU intensity. For example, running an I/O-intensive program fragment at a lower CPU frequency can often save energy without hampering performance (see § 2.2 for details).

Operationally, our approach relies on profiling to assign desirable CPU frequencies to *hot methods*, the methods identified by the JIT for their frequent execution. This design decision is rooted in the fact that hot methods are frequently executed, and any improvement to their energy behavior may have an amplified effect. A fundamental challenge in design is that the gain resulted from DVFS is often eclipsed by the time/energy overhead introduced by DVFS itself. We address this challenge with two solutions. First, we come up with an automated energy profiling process to identify the most energy-consuming hot methods, so that the optimizer can focus more on how "energy hotspot" code regions respond to DVFS. Second, we introduce a form of *counter-based sampling* to DVFS instrumentation, so that the overhead introduced by DVFS is negligible given a reasonable range of sampling rates.

In contrast, the state-of-the-art approach for DVFS-based energy management relies on dynamically monitoring system states, e.g., the rate of cache or TLB misses. A classic example of this approach is the `ONDEMAND` governor, the default power governor in many Linux versions. This governor continuously predicts the level of CPU activities, and adjusts the CPU frequency to meet the demand. This approach is oblivious to the logical structure of the running application, and is fundamentally reactive: it uses the level of CPU intensity at the *current* time interval to set the CPU frequency for the *next* time interval. Whereas the reactive approach is effective when the application is stable within a phase, it loses its effectiveness when there is a phase change. In philosophy, our approach is more aligned with a small body of work that relies on compilers or runtimes to guide DVFS [50, 28, 59, 24, 58]. The relationship between these approaches and ours will be discussed in § 6.

1.2 Contributions

We introduce VINCENT [2], the incarnation of JVM-level method-grained DVFS as an extension to JikesRVM [4, 3]. This paper makes the following contributions:

- the design of a profile-directed energy optimizer, an end-to-end solution that can automatically identify the most energy-consuming hot methods, determine the judicious frequency settings for executing hot methods, and apply DVFS for optimization;
- the specification of method-grained energy optimization at the level of JVM, including the low-overhead sampling algorithm for energy profiling and optimization;
- the implementation and evaluation of method-grained DVFS, which demonstrates its effectiveness relative to existing power governors.

VINCENT is an open-source project. Its source code and all raw experimental data can be found online [3].

2 Background

VINCENT lies at the intersection of two active yet largely independent research directions, energy-efficient computing and managed language runtimes, which we briefly review now.

2.1 Energy Optimization and Metrics

In physics, *energy* (in the unit of joules) is the multiplication of *power* (in the unit of watts) and *time* (in the unit of seconds). Not to lose generality, energy optimization techniques fall into 3 categories: (1) reducing power only; (2) reducing time only; (3) balancing the trade-off between power and time. The first route is an established area of research in hardware design, such as low-power VLSI design [17]. The second route is also mundane: any compiler or runtime optimization that can reduce the execution time of a program can be broadly viewed as an energy optimization. As these first two routes should be more properly named *power* optimization and *performance* optimization respectively, most existing *energy* optimization techniques *de facto* refer to the third route above, which VINCENT also belongs to.

The obvious metric for evaluating energy efficiency is the energy consumption itself. In practice however, as most energy optimization techniques are a balancing act between power and time, the effect of these techniques on power and time should not be ignored. This is particularly true for time, as maintaining performance is an implicit and universal goal. As a result, a prevalent metric for evaluating energy efficiency is the *Energy-Delay Product* (EDP), the multiplication of energy and time. A lower EDP is aligned with our intuition that the energy consumption is reduced while the application remains performant.

2.2 DVFS

DVFS [27, 13] is a classic CPU hardware feature that enables the trade-off exploration between power and time. Except for specialized embedded CPUs, DVFS is supported in nearly all commodity CPUs available today. With DVFS, the operational frequency of a CPU can be dynamically adjusted, such as from 2Ghz to 1Ghz. Strictly speaking, DVFS is a

[2] "I have tried to express the terrible passions of humanity by means of red and green." – Letter from Vincent van Gogh to Theo van Gogh, Arles, 8 September 1888

[3] `https://bitbucket.org/vincent-paper/vinccent`

power optimization design: the power consumption of a CPU has a near cubic relationship with its operational frequency; as a result, when the operational frequency is reduced (or *scaled down*), the power reduction can be dramatic. What makes DVFS a challenging *energy* optimization solution is that, when the CPU frequency is lowered, the execution time of a program typically becomes longer. Recall our earlier discussion that energy consumption is the multiplication of power and time, so the energy consumption effect of DVFS is complex. With EDP as a metric placing more emphasis on time (i.e., not energy consumption alone), the EDP effect of DVFS is even less obvious.

Empirically, downscaling is most effective when the program execution is less dependent on the CPU clock speed. The well known example is the I/O-intensive workload: the program may be waiting for an I/O to complete, and a wait will cause CPU pipeline stalls no matter what frequency is used.

Informally, DVFS is also known as *throttling*. This widely used informal term has an undertone to emphasize the effect of downscaling. Note that DVFS as an approach subsumes both *downscaling* and *upscaling*. The latter refers to the scenario when the operational frequency of the CPU is increased. Upscaling increases power, but may serve as a performance optimization (i.e., reducing execution time).

DVFS, when implemented, takes the form of a system call, where a special system file is written. Each DVFS call generally takes tens of microseconds to complete in modern CPUs [31].

2.3 OS Governors

DVFS provides the hardware capability on adjusting CPU frequencies, but in itself, no algorithm is defined on *when* scaling should happen, and *what* frequency the CPU should be scaled to. The latter is provided through OS-level algorithms called *governors*. The implementation of governors is platform-dependent: the algorithm used by the OS depends on what hardware features are available for power management (beyond DVFS itself).

For generality reasons, Linux provides a set of *generic governors* that do not require additional hardware support [6]. The `ONDEMAND` governor adjusts the underlying CPU frequency based on monitoring the status reported by the performance counters, and a higher CPU frequency is applied when a higher workload is encountered, and *vice versa*. Relative to the middle-of-the-road `ONDEMAND` governor, the `PERFORMANCE` governor on one side of the spectrum is a *time-biased* DVFS regulation algorithm; it lays emphasis on preserving execution time by setting the CPU frequency to be as high as possible. On the other side of the spectrum, the `POWERSAVE` governor is a *power-biased* DVFS regulator, laying more emphasis on reducing power consumption by setting the CPU frequency to be as low as possible. To facilitate customized energy optimization, Linux also comes with a `USERSPACE` governor, deferring all decisions of *when* and *what* decisions of DVFS to the layers of the software stack above the OS.

With additional hardware support for power management, the OS governor can delegate some regulation tasks to the hardware. One example is the Intel P-State [31, 30] support, where the CPU can be set to different power state levels. Instead of operating at a per-core level, the P-State power management operates at the level of a CPU package shared by all cores. When a particular P-State is set, the hardware is able to balance off the individual CPU frequencies of different cores to achieve a particular power budget. More recently, the question of *when* power state transitioning should happen can also be managed by the hardware itself, a feature called hardware-managed P-states (HWP).

On Intel architectures with P-State support, Linux power management can operate in either the *passive* mode or the *active* mode for power management [5]. For architectures without HWP, Linux defaults its behavior to the passive mode, where the Linux generic governors – `ONDEMAND`, `PERFORMANCE`, `POWERSAVE`, and `USERSPACE` – remain in use, except that setting the highest/lowest CPU frequencies in the generic governors are now supported as setting the highest/lowest power states. On Intel architectures with HWP support, Linux defaults its behavior to an *active* mode of P-state use, essentially deferring all its “govenoring” ability to the HWP hardware itself. In the active mode, there is no longer a `USERSPACE` governor; in other words, application-specific or user-specific DVFS is *not* allowed.

2.4 Energy Measurement and RAPL

A relatively independent design and evaluation question is how the energy consumption can be measured. For example, a traditional approach is to rely on the external power/current meters. With the progress of energy-aware computing, newer architectures come with hardware interfaces that can directly query the energy consumption of a computer system “live.”, i.e., during the execution of its hosted application.

The most widely known hardware feature is Intel’s Running Average Power Limit (RAPL) [20], available on all Sandybridge or newer Intel CPUs since 2011 and AMD’s RAPL-compatible CPUs. RAPL can dynamically report the hardware energy consumption and incrementally store it in Machine-Specific Registers (MSRs). The reported energy consumption includes (i) CPU core energy consumption; (ii) CPU uncore energy consumption, i.e., those of on-chip caches, bus controllers, etc; (iii) DRAM energy consumption. RAPL has other features, such as capping the power consumption of a CPU, beyond the scope of this paper.

When implemented, each RAPL reading can be obtained through a number of reads to MSR registers, taking tens of microseconds in modern CPUs. To determine the energy consumption of an execution, a user may take one RAPL reading at the beginning of the execution and the other at the end, and compute the difference of the two.

2.5 JVM Design and JIT

We briefly summarize key aspects of JVM design relevant to this paper. Vincent is built on top of JikesRVM, a representative research-oriented JVM. Research on JikesRVM contributed significantly in JVM design such as on JIT compilation and garbage collection.

JIT compilation allows selected bytecode to be dynamically compiled. One key component of JIT design is to determine which code fragments are most worthy for dynamic compilation. From JikesRVM to HotSpot, a common approach to this task is *hot method selection*: the JVM runtime observes the most frequently encountered methods and select them as the candidate for JIT. Conceptually, the JVM can achieve this task by keeping record of how frequent the beginning (commonly called the *prologue*) and the end (the *epilogue*) of each method are encountered. Realistic JVMs are more sophisticated implementations of this view, for reasons of both improving precision and reducing overhead.

2.6 Counter-Based Sampling

Precisely accounting for the number of times each method is called is expensive. Practical implementations are mostly *sampling*-based: the JVM only counts the prologue/epilogue encounters at time intervals.

Figure 1 Counter-based Sampling.

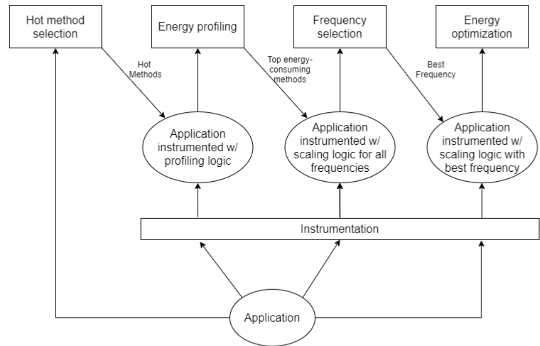


Figure 2 VINCENT Design and Workflow (The top 4 boxes refer to the 4 passes of VINCENT workflow, subsequently from left to right. Each circle represents the application under optimization, in different forms of instrumentation. Each arrow refers to a data dependency/flow).

In JikesRVM for example, a timer thread runs so that a sample is taken at fixed time intervals. JikesRVM further enhances this model by introducing *counter-based* sampling [7], allowing multiple samples to be collected within a time interval. The benefit of counter-based sampling in improving the accuracy of sampling is well documented, especially for complex call graphs where methods are of variant lengths. As shown in Fig. 1, the counter-based approach alternates between taking samples and skipping samples within each time interval. This is achieved through maintaining two counters: the number of samples to take and the number of samples to skip between two samples. VINCENT will adopt JikesRVM's counter-based sampling for *energy* profiling and optimization.

3 Vincent Design

In this section, we describe the design of VINCENT, with a high-level description in § 3.1, followed by an algorithm specification in § 3.2.

3.1 System Overview

A Conceptual Overview

The system components and the workflow of VINCENT are shown in Fig. 2. On the high level, VINCENT is a profile-directed optimizer that conceptually consists of 4 passes:

- **Hot Method Selection**: VINCENT first obtains a list of *hot methods*.
- **Energy Profiling**: VINCENT profiles the energy consumption of hot methods under the default `ONDEMAND` governor. It ranks their energy consumption, and reports a list of *top energy-consuming methods* as the output of this pass.

- **Frequency Selection**: For each top energy-consuming method, VINCENT observes the energy consumption and execution time of the application when the execution of this method is scaled to each CPU frequency, which we call a *configuration*. For each top energy-consuming method, VINCENT ranks the efficiency of its different configurations according to energy metrics, and selects the most efficient one.
- **Energy Optimization**: VINCENT runs the application when the execution of each top energy-consuming method is scaled to the CPU frequency determined in the Frequency Selection phase.

The core design elements are the algorithms for energy profiling (the second pass) and method-based scaling (the third/fourth passes), which we will detail in § 3.2. Conceptually, one may view each pass as a separate run of the application, in the same spirit as a profile-guided optimizer. Therefore, the "energy profiling" pass and the "frequency selection" pass are two separate runs, which we informally call the *profiling run* and the *scaling run*, respectively.

The key observation over this workflow is that VINCENT places the spotlight on *methods*: in each of the workflow tasks, the unit of processing – be it selection, profiling, or optimization – is *at the granularity of methods.*

A High-Level Implementation Overview

From the implementation perspective, VINCENT builds on top of JikesRVM, and we resort to existing support in JikesRVM for the first pass, Hot Method Selection. JikesRVM's built-in process–from how to sample methods to what heuristics are introduced to determine hotness–is not altered. Conceptually, hot method selection can be a separate run of the application itself, outputing a list of methods that JikesRVM deems "hot." In our implementation, the hot method selection and profiling is combined in one run: i.e., whenever a hot method is identified during the execution of an application, the energy profiling component of VINCENT will start profiling its energy consumption. In this regard, the VINCENT development interfaces with existing JikesRVM logic through a common data structure where hot methods are kept: whenever such a data structure is updated by JikesRVM, VINCENT under the profiling run will start profiling for the newly added entry. We also follow a similar implementation for the scaling run.

In addition, VINCENT does not alter the dynamic compilation process of JikesRVM, except that the additional logic for profiling (or scaling) is inserted through instrumentation at the beginning of the dynamic compilation process. Take the profiling run for instance. Whenever a hot method is identified, we dynamically instrument that method with the VINCENT profiling logic in the profiling run, which will be subsequently compiled by JIT dynamic compilation.

3.2 Vincent Specification

We now specify the algorithm implemented by VINCENT. We first describe the top-level thread bookkeeping (§ 3.2.1), and then the profiling algorithm (§ 3.2.2) and the scaling algorithm (§ 3.2.3).

3.2.1 Thread Bookkeeping

Algorithm 1 overviews the bookkeeping in a multi-threading environment. Here, all threads visible to the JVM (other than the timer thread itself) are maintained in a global structure *ts*, a collection of threads of type T. Each thread contains thread-local bookkeeping information;

Algorithm 1 Thread Bookkeeping and Timer Thread Loop.

```
1: typedef T {
2:     vtimer: int // timer
3:     skipCount: int // # calls to skip
4:     sampleCount: int // # samples to collect
5:     edata: EDATA // energy profiling data
6:     gov: GOVERNOR // saved governor
7:     freq: FREQ // saved CPU frequency
8: }
9: const EPOCH // time unit
10: const SKIPNUM // skipped samples between
11: const SAMPLENUM // samples per interval
```

```
1: ts: T[THREADNUM] // running threads
2: procedure TIMER
3:     while TRUE do
4:         SLEEP(EPOCH)
5:         for each t ∈ ts do
6:             t.vtimer++
7:         end for
8:     end while
9: end procedure
```

in particular, note that *vtimer* manages the elapse of time, incremented by the unit EPOCH. As profiling and scaling belong to different passes of VINCENT and do not share the same runtime, *vtimer* is used for both runs. The thread-local fields used only for profiling and those only for scaling are illustrated with GREEN box and LIME box respectively. The specific meanings of the constants and the fields in T other than *vtimer* will be detailed in the rest of this section.

The timer thread is defined as an infinite loop. When the JVM timer interrupt happens at the rate of EPOCH, the *vtimer* associated with each thread is incremented.

In the rest of this section, we specify our algorithm design for energy profiling and DVFS-based energy optimization. Both passes are unified by one fundamental hurdle: if naive instrumentation is used, the overhead for obtaining raw energy samples (in energy profiling) and the overhead for performing DVFS (in energy optimization) are too high. We now detail our solution in § 3.2.2, i.e., how we overcome the overhead challenge of obtaining raw energy samples in energy profiling through a sampling-based approach. Note that in § 3.2.3, the same sampling-based solution is also used for DVFS-based energy optimization to overcome the challenge posed by the overhead for performing DVFS.

3.2.2 Profiling Instrumentation

Recall that the goal of profiling is to identify the top energy-consuming methods. The raw energy consumption maintained by the underlying hardware (see § 4) is *accumulative*, i.e., reported as monotonically increasing values. To determine the energy consumption of a method, we conceptually need to "diff" the raw energy reading obtained at the beginning of the method execution, and one obtained at the end of the method execution.

3.2.2.1 Challenges and Strawman Solutions

Obtaining a raw energy reading from the underlying hardware incurs a non-trivial overhead, often taking tens of microseconds to complete. As a result, standard solutions known to be effective for execution time profiling may not be ideal for energy profiling, which we now briefly review.

A strawman solution naively adapted from execution time profiling is to instrument the begin (i.e., prologue) and the end (i.e., epilogue) of every hot method, where a raw energy reading is taken each time the prologue and epilogue is encountered. The energy consumption of a method can thus be the difference between the two readings. Unfortunately,

Algorithm 2 Profiling Algorithm.

```
 1: typedef LOG {
 2:   mn: MNAME // method name
 3:   edata: EDATA // data
 4: }
 5: typedef CVAL enum { TAKE, SKIP, LAST }
 6: typedef EDATA float
 7: const PN // profiling timer factor
 8: l: LOG[LOGNUM]

 9: procedure PROLOGUEPROFILE()
10:     t ← CURRENTTHREAD()
11:     if COUNTER(t, PN) == TAKE then
12:         t.edata ← READENERGY()
13:     end if
14:     if COUNTER(t, PN) == LAST then
15:         t.edata ← ⊥
16:     end if
17: end procedure

18: procedure EPILOGUEPROFILE()
19:     t ← CURRENTTHREAD()
20:     if COUNTER(t, PN) == TAKE or LAST then
21:         e ← READENERGY()
22:         if t.edata ≠ ⊥ then
23:             l +← LOG(THISM, DIFF(e, t.edata))
24:         else
25:             t.edata ← e
26:         end if
27:     end if
28:     if COUNTER(t, PN) == LAST then
29:         t.edata ← ⊥
30:     end if
31: end procedure

32: function COUNTER(t: T, factor: int): CVAL
33:     if t.vtimer >= factor then
34:         t.skipCount ← t.skipCount − 1
35:         if t.skipCount == 0 then
36:             t.skipCount ← SKIPNUM
37:             t.sampleCount ← t.sampleCount − 1
38:             if t.sampleCount == 0 then
39:                 t.vtimer ← 0
40:                 t.sampleCount ← SAMPLENUM
41:                 return LAST
42:             end if
43:             return TAKE
44:         end if
45:     end if
46:     return SKIP
47: end function
```

thanks to the non-trivial overhead with RAPL energy readings, this approach may incur prohibitively high overhead (10x-200x in our preliminary experiments), severely altering the program behavior. In other words, the instrumented run may produce the result no longer representative of the original benchmark's energy behavior. Observe that even instrumenting each hot method "one at a time" does not solve the problem. The hot methods are "hot" for a reason: they are frequently called, and the per-call overhead may rapidly accumulate.

A second strawman solution is to perform sampling at fixed time intervals. For example, assume the JVM has just taken an energy sample of $90J$ at the beginning of its 100th time interval. After one time interval elapses, it takes another energy sample of $90.25J$, and the epilogue of a method is encountered. The approach can thus attribute $0.25J$ to that method. This approach however may lead to over-attribution: $0.25J$ is attributed to one method encountered at the end of the time interval, but many other methods may have contributed to the energy consumption during the interval. This sampling approach is widely used for execution time sampling, because precision can be improved by shortening the time interval. For energy profiling however, the room for shortening the time interval is limited due to the overhead of raw energy readings.

3.2.2.2 Delimited and Counter-Based Sampling with Vincent

To address these challenges, the solution adopted by VINCENT consists of two ideas: *delimited sampling* and *counter-based sampling*. Overall, the former is an overhead-reducing approximation that combines the strawman solutions above, and the latter is a precision-increasing optimization over the general sampling-based approach.

Delimited Sampling. The energy profiler of VINCENT is a hybrid of the two strawman solutions above, which we call *delimited sampling*. Similar to the first strawman approach, VINCENT takes energy readings when the method prologue and the method epilogue are encountered, and computes the difference of the two. VINCENT however does not take energy readings at every encounter of the prologue or the epilogue. Instead, the number of energy readings at the method prologue/epilogue are *bounded for each interval*, similar to the second strawman approach.

As seen in Algorithm 2, each hot method is instrumented with a pair of methods, with PROLOGUEPROFILE inserted before the entry point of the method body, and EPILOGUEPROFILE inserted after each exit point of the method body. Auxiliary function READENERGY obtains a raw energy sample from the underlying hardware (a value of EDATA type). Binary function DIFF computes the difference of two raw energy samples, and function CURRENTTHREAD returns the current thread of the execution, of type `T`. Constant `THISM` is the name of the instrumented method, an implementation detail we clarify in § 4. Sampling happens within the function of COUNTER, which we will describe shortly.

The key observation here is that we are not attempting to replicate the first strawman approach, but to avoid the overattribution problem in the second strawman approach. The philosophy here is *refutation*: if a prologue or epilogue (of any method) is encountered before the epilogue of the method m of our interest, we know the energy consumption incurred before the prologue encounter must not be due to m, thanks to how call stacks are structured. This can be concretely observed in the specification of EPILOGUEPROFILE in Algorithm 2. At Line 23, the energy difference between a prior energy sample and the current energy sample is computed. Now that the method has reached its epilogue, the "current energy sample" intuitively keeps the accumulated energy value until the method reaches its end. The intriguing question however is when the "prior energy sample" is collected. Delimited sampling introduces an approximation: it is collected during the last time in the sampling trace when a method is called (i.e., a prologue is executed) or a method is returned (i.e., an epilogue is executed). They can be seen at Line 12 and Line 21 respectively in Algorithm 2. In other words, the refutation-based algorithm says that any prior encountered prologue or epilogue "*delimits*" where the method could start: any energy consumption before the last method is called or returned *must not* belong to the current method we encounter in the epilogue.

On a more technical level, treating the prior encounter of an epilogue as a "limit" of the method start (as well as the prior encounter of a prologue) is also friendly for accounting for the energy consumption of a recursive/nested method. For some applications, the hot method happens to be a recursive call. When a sample is ready to be taken, it is possible that the activation record of the recursive call is popping. Without Line 21, the sampling algorithm would only take the next energy sample when a prologue is executed (i.e., a push), and hence would miss a round of sampling in this pop-only phase of recursive execution. With Line 21, the energy consumption between 2 pops can be recorded and attributed to the recursive method.

Finally, note that the energy accounting specified here is conceptually "flat": in the presence of a call chain where both the caller method and the callee method are hot, the callee's energy consumption is *not* accounted as a part of the caller's energy consumption. This is implied in the delimited approach itself: when the epilogue of the caller method is encountered, the epilogue of the callee method is already encountered. As a result, only the energy consumption after the callee method is completed is attributed to the caller method. Indeed, due to sampling, our implementation is an approximation of this conceptually flat view.

Algorithm 3 Scaling Algorithm.

```
 1: enum GOVERNOR {USERSPACE, ONDEMAND, ...}
 2: const SN // scaling timer factor

 3: procedure PROLOGUESCALE(f : FREQ)
 4:     t ← CURRENTTHREAD()
 5:     if COUNTER(t, SN) == TAKE then
 6:         t.gov ← GETGOVERNOR()
 7:         if t.gov == USERSPACE then
 8:             t.freq ← GETFREQ()
 9:         else
10:             SETGOVERNOR(USERSPACE)
11:         end if
12:         SETFREQ(f)
13:     end if
14:     if COUNTER(t, SN) == LAST then
15:         SETGOVERNOR(ONDEMAND)
16:     end if
17: end procedure

18: procedure EPILOGUESCALE()
19:     t ← CURRENTTHREAD()
20:     if COUNTER(t, SN) == TAKE then
21:         if t.gov ≠ ⊥ then
22:             SETGOVERNOR(t.gov)
23:             if t.gov == USERSPACE then
24:                 SETFREQ(t.freq)
25:             end if
26:         end if
27:     end if
28:     if COUNTER(t, SN) == LAST then
29:         SETGOVERNOR(ONDEMAND)
30:     end if
31: end procedure
```

Counter-based Sampling. Our description so far can be *conceptually* viewed as taking two energy readings – one at the prologue and the other at the epilogue – for each time interval. VINCENT extends from this conceptual view by adopting counter-based sampling (see § 2), allowing multiple (but still bounded) pairs of energy readings to be collected within a time interval. In general, counter-based sampling is a precision-improving strategy known to strike a balance for accounting both long methods and short methods. Specific to energy optimization, this means that VINCENT cares about both longer but slightly less frequently invoked (but still hot) methods and shorter but more frequently invoked methods, as long as they incur high energy consumption.

In Algorithm 2, counter-based sampling is captured by function COUNTER, at Lines 32-47. Here, the profiling time interval is set as PN $\times$ EPOCH; recall that *vtimer* is incremented at each VM EPOCH, so PN is the "slowdown" factor of profiling relative to the top-level timer loop. Constants SAMPLENUM and SKIPNUM represent the number of samples to take and skip, respectively, within each profiling time interval.

The COUNTER function may return one of the 3 values: TAKE (indicating a sample should be taken), SKIP (indicating a sample should not be taken), and LAST (indicating one last sample should be taken for each time interval). The LAST value plays a role of re-initializing the environment for the next time interval. For profiling, this means to reset the *edata* field.

Finally, observe that the COUNTER function only accesses data that records the state of the *current* thread. This can be observed that every access in this function is prefixed with variable *t*. In other words, it is not possible for two application threads to access the same fields in a race condition.

3.2.3 Scaling Instrumentation

Algorithm 3 defines the instrumentation-based algorithm for CPU scaling. Convenience function GETGOVERNOR retrieves the current governor (power manager) from the underlying system, which can either be USERSPACE (i.e., with frequencies manually set by the user) or ONDEMAND. Function SETGOVERNOR sets the governor to its argument value. Function GETFREQ retrieves the current CPU frequency, whereas SETFREQ sets the CPU frequency to its argument value.

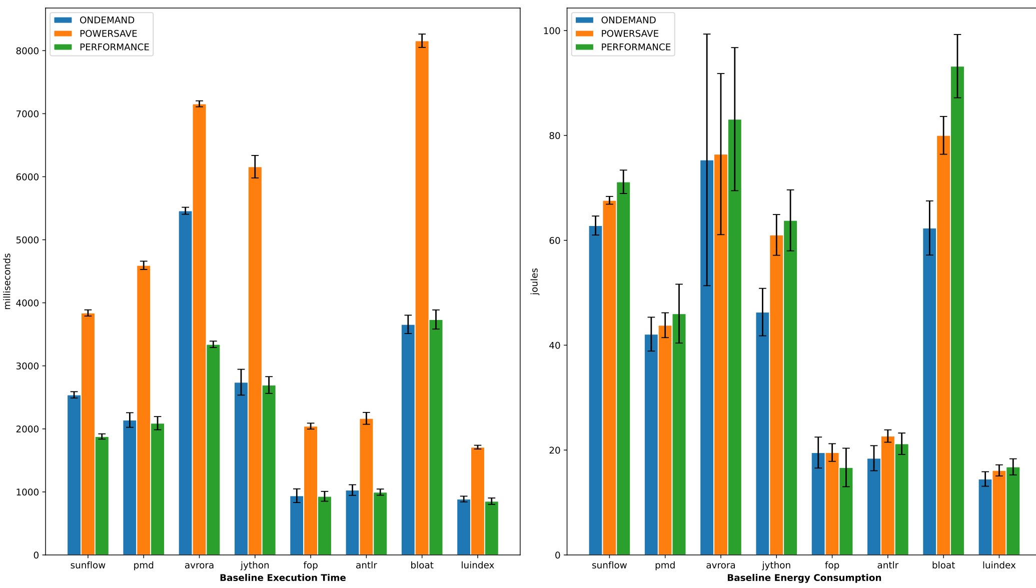


Figure 3 Benchmark Statistics under Different Governors as Evaluation Baselines.

Recall that the scaling instrumentation is used for VINCENT's passes of frequency selection or energy optimization. The instrumentation is only applied to the hot top-energy consuming methods. When the application is bootstrapped, VINCENT sets the governor to `ONDEMAND`. When a top energy-consuming method is encountered at its PROLOGUESCALE, the governor and the CPU frequency are set according to the need of frequency selection or energy optimization. At this point, the governor to be used is `USERSPACE`, *a la* the convention of Linux. VINCENT in addition preserves the governor/frequency context, i.e., the settings of governor/frequency before the PROLOGUESCALE is encountered. The EPILOGUESCALE recovers the preserved context.

Just as profiling, counter-based sampling is also at work during scaling. Note that profiling and scaling do not have to follow the same rate. Constant `SN` adjusts the rate for scaling. In addition, note that when we reach the `LAST` sample in each time interval, the governor is reset to `ONDEMAND`.

4 Implementation and Experimental Settings

4.1 Hardware/OS/VM Setup

We evaluated VINCENT on a dual socket Intel E5-2630 v4 2.20 GHz CPU server, with 10 cores in each socket and 64 DDR4 RAM. Hyperthreading is enabled. In total, we have 20 physical cores and 40 virtual cores. The machine runs Debian 9.11 (stretch), Linux kernel 4.9. For profiling based on individual CPU frequencies and the DVFS-based optimization, we explored all CPU frequencies that can be stably supported by our hardware, ranging from 2.2GHz to 1.2GHz, with the decrement of 0.1Ghz. For the rest of the paper, we use `F1` to refer to 2.2Ghz, `F2` for 2.1Ghz, `F3` for 2.0Ghz, ..., `F11` for 1.2Ghz. The CPU frequencies are switched through the `scaling_setspeed` file, under the directory of `/sys/devices/system/cpu/cpu*/cpufreq` for CPU cores.

Intel E5-2630 v4 is an instance of the Intel Broadwell architecture. It supports P-states but does not have HWP support. The P-states operate in the `passive` mode (see § 2), and the Linux governors of `ONDEMAND`, `PERFORMANCE`, `POWERSAVE`, `USERSPACE` remain available. The governors are switched through setting the `scaling_governor` file under the same directory as above. Recall that the `active` mode does not support `USERSPACE` govenor, so it cannot be used for VINCENT. To avoid feature intervention, Turbo boost is turned off. None of the experiments described in this paper (including both for VINCENT and for baselines) alters other system settings related to power management.

We rely a Java-based tool jRAPL [38] to obtain raw RAPL energy readings. The energy consumption reported by RAPL is accumulative. Each energy sample – as shown of the `EDATA` type in the algorithm specification – is the sum of energy readings from all sockets; and each socket-wise reading consists of energy consumption for the CPU cores, the uncore (cache, TLB, etc), and the DRAM. Specific to our environment, this means we collect and sum up $2 \times 3 = 6$ raw readings for each energy sample.

We implemented VINCENT on JikesRVM version 3.1.4. The hot method selection is built on top of the Adaptive Optimization System (AOS) [8] of JikesRVM.

4.2 Hot Method Selection

We rely on the JIT component of JikesRVM for hot method selection. We do not alter JikesRVM's hot method selection logic. The interaction between the JikesRVM logic and VINCENT is primarily through the data structure where hot methods are placed: whereas JikesRVM places hot methods into the structure, the profiling/scaling logic of VINCENT reads from it. The hot method selection process in JikesRVM is adaptive, so is the process of profiling based on them. Whenever a new method is identified as hot, VINCENT's profiler will instrument it dynamically and perform its profiling upon identification.

One design consideration was whether we should exclude very short methods such as getters and setters from the hot methods. Intuitively, if such methods were subjected to scaling, the scaling overhead might well offset the benefit of setting the method to the desired frequency. Fortunately, the top energy-consuming methods identified by VINCENT's energy profiler (as seen in § 5) appear to rarely include them. In other words, these very short methods, even though hot from the perspective of invocation counts, rarely accumulate enough energy consumption to become top energy-consuming methods. As a result, we choose to keep our design simple, and do not alter the hot method selection logic in JikesRVM.

4.3 Algorithm Implementation

The prologue and epilogue program fragments for profiling and optimization we specified in the previous section are inserted as IR instrumentation through `hir2lir`. Recall that we need to obtain the "this method" information (`THISM` in Algorithm 2). This is implemented through instrumentation: as the method signature is carried with the IR, VINCENT stores the method information when instrumentation is added. Other than this instrumentation, we preserve the original JikesRVM logic for dynamic compilation.

In the top-level timer loop, the interval `EPOCH` is identical to the default time interval of AOS, 4ms. Unless otherwise noted, we set the time interval for both profiling and scaling at 8ms, i.e., `PN` = 2 and `SN` = 2. Within each time interval, counter-based sampling is at work for both profiling and scaling. Unless otherwise noted, parameter `SAMPLENUM` is set at 16. In both scenarios, `SKIPNUM` = 7. The fact the skipped number of samples should be an odd number is well known in counter-based sampling [7].

All energy readings are stored as a C array and printed after the experiments end for posterior analysis.

4.4 Benchmarking and Experimental Setup

We evaluate VINCENT with benchmarks in the Dacapo suite [12], arguably the most widely used benchmark suite for multithreaded Java applications. Our benchmarks by default come from the last version of Dacapo known to work with JikesRVM, Dacapo MR2. Dacapo has a more recent release, Dacapo 9.12-bach, and we successfully ported some benchmarks in this version – `sunflow`, `luindex`, and `avrora` specifically – to work with JikesRVM. The rest of porting was unsuccessful because JikesRVM cannot support some advanced Java features that appeared in the later versions of benchmarks.

4.5 Baselines

To evaluate the effectiveness of VINCENT, we choose 3 baselines. They are the three application execution scenarios where DVFS is guided by the `ONDEMAND`, `POWERSAVE`, and `PERFORMANCE` OS governors respectively (see § 2). They are representative scenarios of running Java applications on commodity software/hardware stack today. As variants of DVFS approaches guided by dynamic monitoring, they set a contrast with the core idea of VINCENT's approach, *method*-based DVFS.

The baseline execution time and energy consumption of each benchmark while running with the 3 Linux governors can be found in Fig. 3. In addition to serving as experimental baselines, this figure may also help gain intuition on the characteristics of DVFS guided by the 3 governors. For example, the `PERFORMANCE` governor often leads to the shortest execution time, as shown in the left sub-figure; it however generally increases the energy consumption, as shown in As shown in the right sub-figure. Overall, the `ONDEMAND` governor strikes a good balance between maximizing energy savings while delivering competitive performance. As a result, we will conduct a more detailed comparative analysis between VINCENT and the `ONDEMAND` baseline in the following section.

Unless otherwise noted, all experiment results throughout the paper (including both baseline runs and VINCENT runs) are collected by running each benchmark 20 times in a hot run, and reporting the average of the last 15 runs.

5 Vincent Evaluation

In this section, we evaluate the effectiveness of VINCENT. We aim at answering the following questions: (**Q1**) Do method-frequency configurations exist that can lead to energy savings and favorable EDPs, compared with existing Linux power governors? (**Q2**) How does the choice of sampling settings impact the effectiveness of VINCENT? (**Q3**) How is VINCENT compared against different existing power management strategies? We answer each of these questions in each subsection below.

5.1 Method-Grained Energy Optimization

5.1.1 Energy Profiling

The VINCENT lifecycle starts with energy profiling. Fig. 4 shows the top-5 energy-consuming methods for selected benchmarks. Thanks to sampling, the reported percentage of energy consumption for each listed method is likely to be lower than its actual normalized energy

sunflow

Rank	Method Name	Percentage(%)
1	`org.sunflow.core.light.TriangleMeshLight.getRadiance`	9.36
2	`org.sunflow.core.primitive.TriangleMesh.init`	4.60
3	`org.sunflow.math.Matrix4.transformP`	2.19
4	`org.sunflow.core.shader.MirrorShader.getRadiance`	0.45
5	`org.sunflow.core.accel.KDTree.BuildTask.<init>`	0.005

pmd

Rank	Method Name	Percentage(%)
1	`org.jaxen.expr.DefaultAllNodeStep.matches`	15.52
2	`org.jaxen.expr.iter.IterableChildAxis.supportsNamedAccess`	8.21
3	`org.jaxen.QualifiedName.hashCode`	7.01
4	`net.sourceforge.pmd.jaxen.DocumentNavigator.getAttributeName`	4.78
5	`org.jaxen.util.SingleObjectIterator.hasNext`	4.18

antlr

Rank	Method Name	Percentage(%)
1	`antlr.CodeGenerator._println`	5.56
2	`antlr.SimpleTokenManager.getTokenSymbol`	5.23
3	`antlr.LLkAnalyzer.look`	3.92
4	`antlr.CSharpCharFormatter.escapeChar`	2.61
5	`antlr.Grammar.getSymbol`	2.61

Figure 4 Top Energy-Consuming Methods According to VINCENT Energy Profiling (The first column is the rank; the second column is the name of the method; the third column is its normalized energy consumption relative to the overall energy consumption of the benchmark).

consumption, but what matters here is the relative standing of the methods: we are able to identify the most-energy consuming methods so that the methods that DVFS should be applied upon are identified.

Very short methods rarely appear in the top energy-consuming methods. One example is `pmd`'s top-consuming method, `DefaultAllNodeStep.matches`, which only contains a simple boolean return as its method body. As we shall see soon, these methods are indeed unfriendly for DVFS (see § 4). That being said, the vast majority of methods identified by VINCENT's profiling phase are methods of reasonable length (in terms of execution time) where the DVFS time overhead is relatively small to the execution time of the method itself.

For VINCENT, the energy profiling results are intermediate. The effectiveness of identifying top energy-consuming methods will impact the effectiveness of energy optimization, which we describe next.

5.1.2 The Impact on Energy Consumption

We now describe the effectiveness of VINCENT energy optimization against the `ONDEMAND` baseline, i.e., when the application is running with the `ONDEMAND` governor in place throughout its execution. We show the energy consumption results of VINCENT in Fig. 5 when a single hot method is scaled to a particular CPU frequency. In each figure, a heat map is used for each

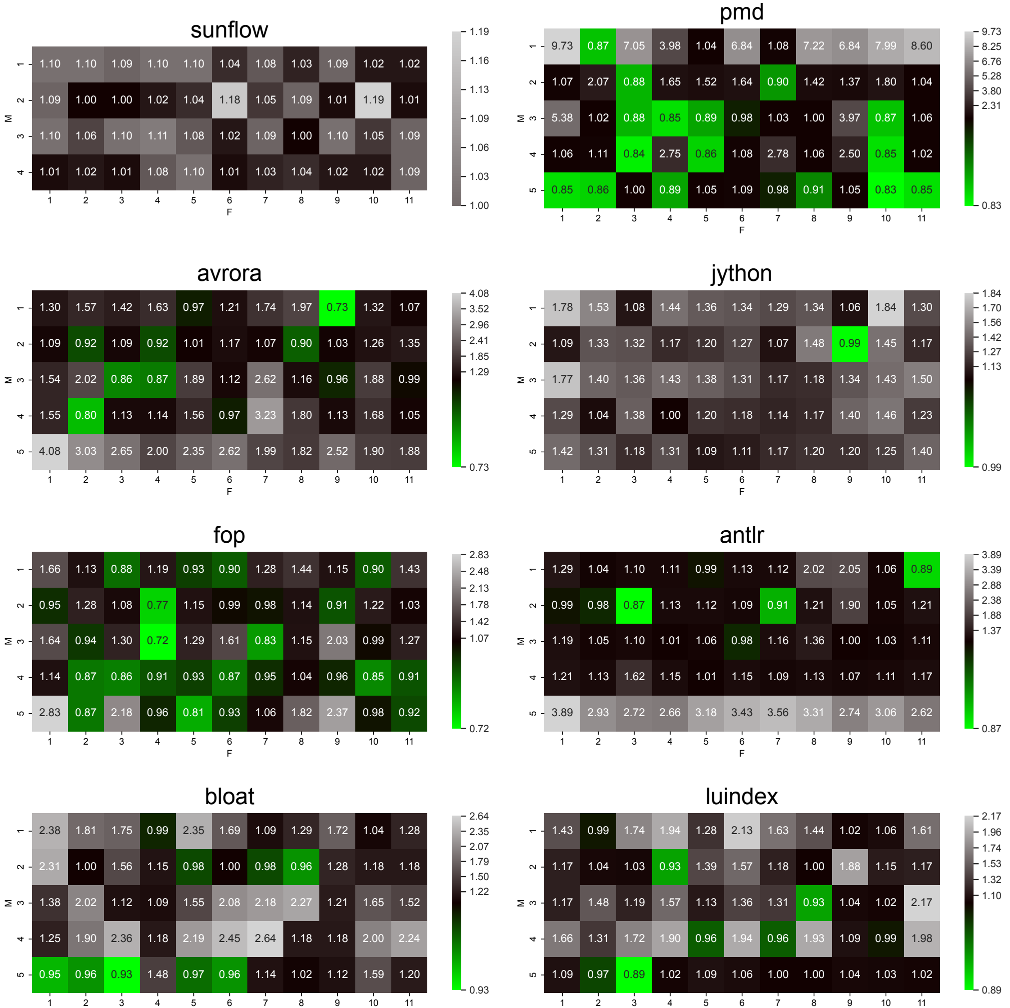


Figure 5 VINCENT Energy Consumption Normalized Against the `ONDEMAND` Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has energy consumption v, normalized against that of the `ONDEMAND` run. If $v < 1$, the VINCENT incurs less energy than the `ONDEMAND` run).

benchmark to show the result of running it with VINCENT where one of the top-consuming methods (Y axis) is subjected to DVFS at a particular frequency level (X axis). The value carried in each cell in the heatmap is normalized against the `ONDEMAND` run. Each green cell indicates an energy-friendly configuration, i.e., the energy consumption for VINCENT is smaller than that of the `ONDEMAND` run. All benchmarks are shown with 5 top-consuming methods except `sunflow`, which we only show 4 because the 5th energy-consuming method consumes little energy, as shown in Fig. 4.

Method-grained energy optimization is effective in reducing energy consumption for all benchmarks (but one): there exists at least one configuration within the benchmark whose normalized energy consumption is less than 1. For example, when VINCENT runs `antlr` at the third highest CPU frequency (2.0Ghz) for its second most energy-consuming method, `SimpleTokenManager.getTokenSymbol`, the normalized EDP is 0.87, indicating

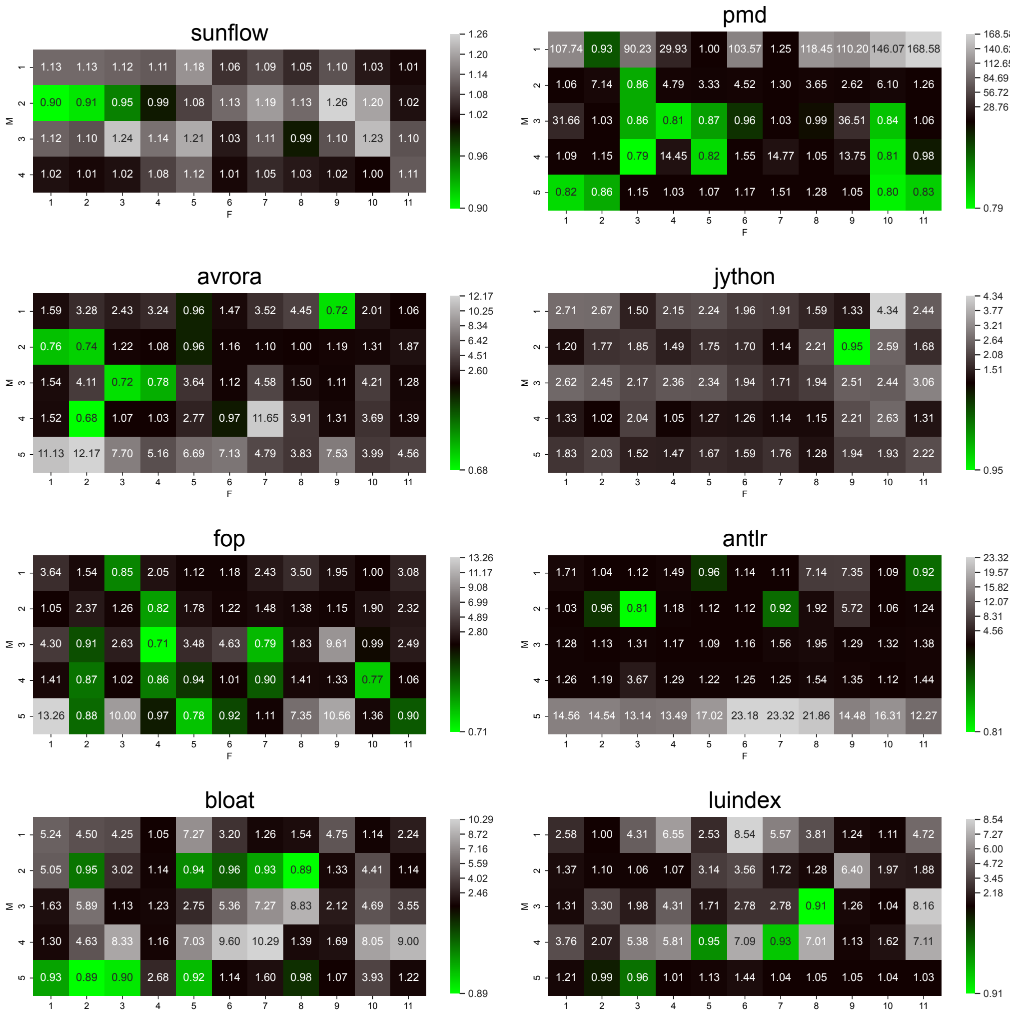


Figure 6 VINCENT EDP Normalized Against the ONDEMAND Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has EDP v, normalized against that of the ONDEMAND run. If $v < 1$, the VINCENT is more energy-efficient than the ONDEMAND run w.r.t. EDP).

that VINCENT can save energy by 13% than running `antlr` with the ONDEMAND governor. As each green cell in the heatmap indicates a configuration with energy savings relative to ONDEMAND, energy optimization opportunities widely exist across benchmarks.

Indeed, not every benchmark can benefit from method-grained energy optimization. Benchmark `sunflow` has all normalized energy consumption values greater than 1 for all VINCENT configurations, indicating the ONDEMAND execution indeed consumes less energy than VINCENT. The same applies to nearly all `jython` configurations. Both benchmarks are consistently CPU-intensive, meaning that the ONDEMAND governor is likely to operate the CPUs at the highest frequencies at most times. In this case, DVFS has limited choices: if it scales the CPU down, the CPU-intensive application may run significantly slower, negatively impacting energy consumption because the latter is the accumulated power consumption *over time*; if it scales the CPU up, the power consumption may increase, ultimately impacting the energy consumption as well.

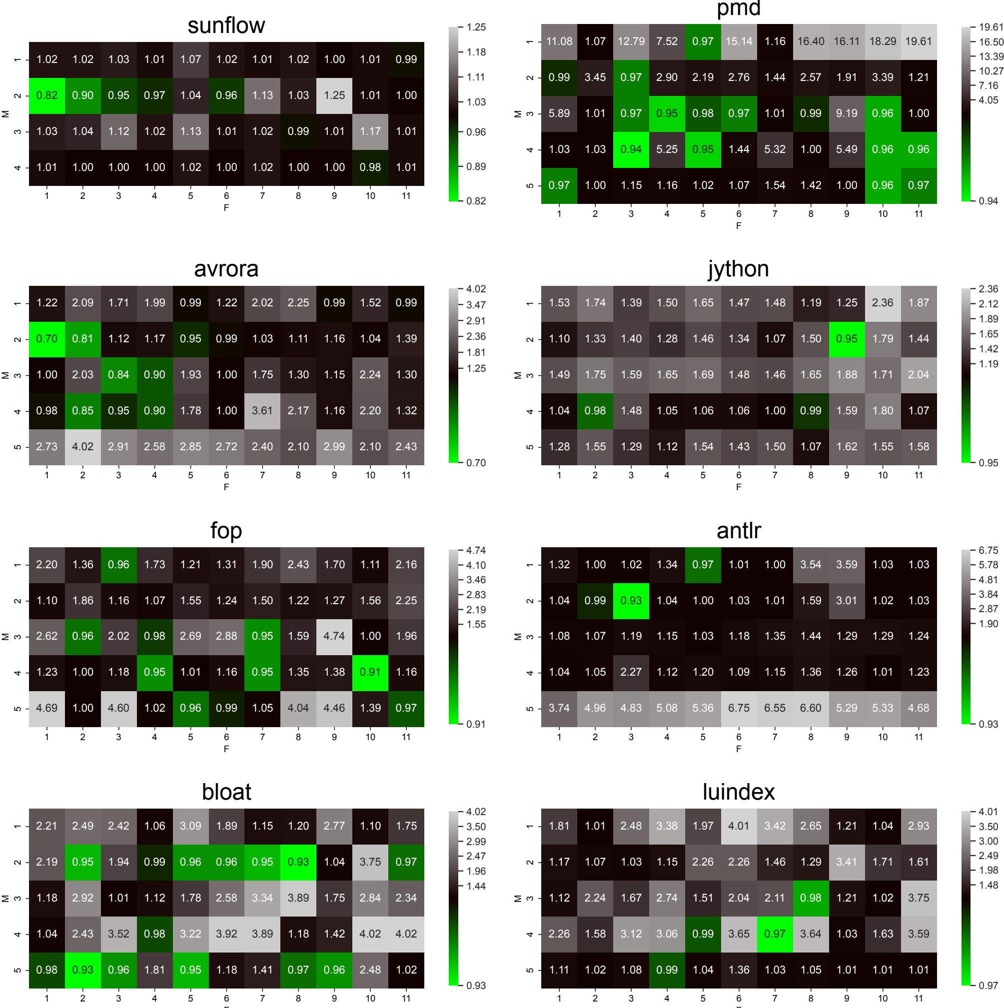


Figure 7 VINCENT Execution Time Normalized Against the `ONDEMAND` Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has execution time v, normalized against that of the `ONDEMAND` run. If $v < 1$, the VINCENT runs faster than the `ONDEMAND` run).

In contrast, memory-intensive or I/O-intensive benchmarks respond well with VINCENT. This is consistent with our general understanding of DVFS: these benchmarks often have latency due to memory round-trips or I/O requests, and scaling down the CPU frequency may have limited impact on execution time while reducing the power consumption significantly. For example, there are benefits for reducing energy consumption for many configurations of `pmd` (AST-based program analysis), `avrora` (simulation), `fop` (file transformation), and `luindex` (data indexing). All are centric to data processing, and most benchmarks have I/Os.

Finally, relatively short methods (such as the top-consuming method of `pmd` and `bloat`) indeed respond to DVFS poorly: the overhead of DVFS significantly outweighs its benefit. As we can see, energy consumption may deteriorate significantly for them, sometimes near 10x.

5.1.3 The Impact on EDP

Fig. 6 shows Vincent's impact on energy consumption. One interesting observation is that DVFS may play different roles for different benchmarks in balancing the trade-off between energy consumption and execution time: sometimes the reduction of EDP is due to reduced energy consumption, whereas at other times, EDP may reduce due to reduced execution time.

Take `sunflow` for instance. Recall earlier that its energy heatmap revealed that reducing the energy consumption of `sunflow` is challenging (all cells in the energy consumption heatmap are red), but observe that Vincent may in fact improve the energy efficiency of `sunflow` in terms of EDP: by scaling the CPU frequency to the highest while executing its method `TriangleMesh.init`, the normalized EDP may reach 0.90, i.e., a 10% reduction than that of `ONDEMAND`. Here, Vincent primarily plays the role of improving the performance: as `sunflow` is a CPU-intensive benchmark, DVFS plays the role of speeding up its execution; the shortened execution time contributes to the reduced EDP.

Overall, we find Vincent an effective solution to reducing EDP as well as energy consumption. Occasionally, it is even more effective for the former than the latter: when we correlate Fig. 5 and Fig. 6, the best configuration for a benchmark often exhibits a lower normalized value in Fig. 6 than in Fig. 5. As energy optimization is a known trade-off between maximizing energy savings and minimizing performance loss, an EDP-friendly solution is of practical importance.

5.1.4 The Impact on Execution Time

In Fig. 7, we show the impact of Vincent on execution time. Observe that every benchmark consists of at least one configuration that may speed up the benchmark relative to its `ONDEMAND` run. At the first glance, the fact that Vincent may serve as a *performance* optimizer may come as a surprise, but this is indeed natural for two reasons.

First, even though DVFS is better known for its effect on energy savings with *downscaling*, the opposite is also true: it can speed up the program execution with *upscaling*. What this figure shows is that Vincent may select a performance-sensitive method and execute it on a higher CPU frequency than an `ONDEMAND` governor baseline would, potentially speeding up the program.

Second, note that `ONDEMAND` governor is a "middle-of-the-road" governor (see § 2) in terms of how aggressive/conservative it scales up CPU frequencies in the presence of workload increase. As we shall see in § 5.2, the `PERFORMANCE` governor is a more challenging baseline to overcome in terms of viewing Vincent as a performance optimization.

5.2 Alternative Baselines

We have so far compared our results with the `ONDEMAND` governor, arguably the most widely used DVFS-enabled energy optimization based on dynamic monitoring. In this section, we now look at other important governors as baselines.

In Fig. 8, we show the relative effectiveness of Vincent against alternative governors. For example, the height of `sunflow` EDP bar against the `ONDEMAND` governor is 0.86, meaning that among all CPU frequencies, all selected methods, and all sampling rate settings, the Vincent configuration with the least EDP is 14% less than that of the `ONDEMAND` run for `sunflow`. For the same benchmark, its EDP bar against the `POWERSAVE` governor is 0.52, meaning that the Vincent configuration with the least EDP is 48% less than that of the `POWERSAVE` run. In other words, `POWERSAVE` is a relatively less effective power governor for `sunflow` than `ONDEMAND` in terms of EDP, and neither is as effective as Vincent.

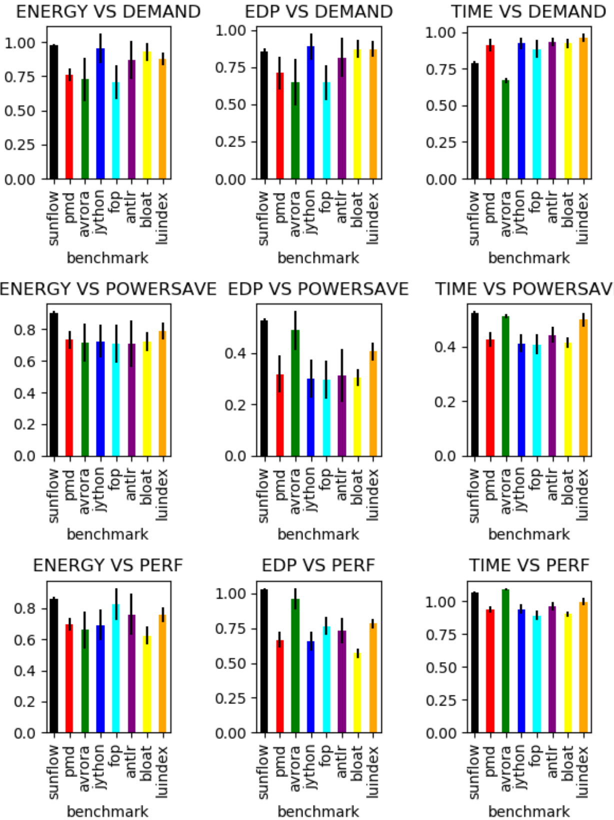


Figure 8 VINCENT Best Results against Different Governor Baselines (The first row shows results normalized against the `ONDEMAND` governor. The second row shows results normalized against the `POWERSAVE` governor. The third row shows results normalized against the `PERFORMANCE` governor. For all bars, shorter is better).

Across the benchmarks, a trend is that the `POWERSAVE` baseline fares poorly relative to `ONDEMAND`, and much worse than VINCENT. Relatively, `POWERSAVE` is slightly worse than the `ONDEMAND` governor in terms of energy consumption, but it may significantly increase the execution time of benchmarks, ultimately leading to poor EDPs.

VINCENT is also more energy-efficient than the `PERFORMANCE` governor. Note that in the last row of Fig. 8, all normalized energy results are significantly less than 1. All but one (`sunflow`) benchmarks also have EDP results less than 1. The most revealing fact about the `PERFORMANCE` governor is that it may reduce the execution time of CPU-intensive benchmarks. Recall that when VINCENT is compared against the `ONDEMAND` governor in terms of the execution time (the last figure in the first row), the VINCENT runs of `sunflow` and `jython` can lead to shorter execution time than the runs with the `ONDEMAND` governor. This however is not true when VINCENT is compared against the `PERFORMANCE` governor: the VINCENT runs of `sunflow` and `jython` are slightly slower than the runs with the `PERFORMANCE` governor (the last figure in the last row). The `PERFORMANCE` governor however is not as effective for memory-intensive or I/O-intensive benchmarks.

The surprising fact is that the VINCENT runs for some benchmarks can in fact lead to a small but noticeable reduction in the execution time than their counterpart `PERFORMANCE` runs. When the `PERFORMANCE` governor is used to regulate DVFS on Intel architectures where

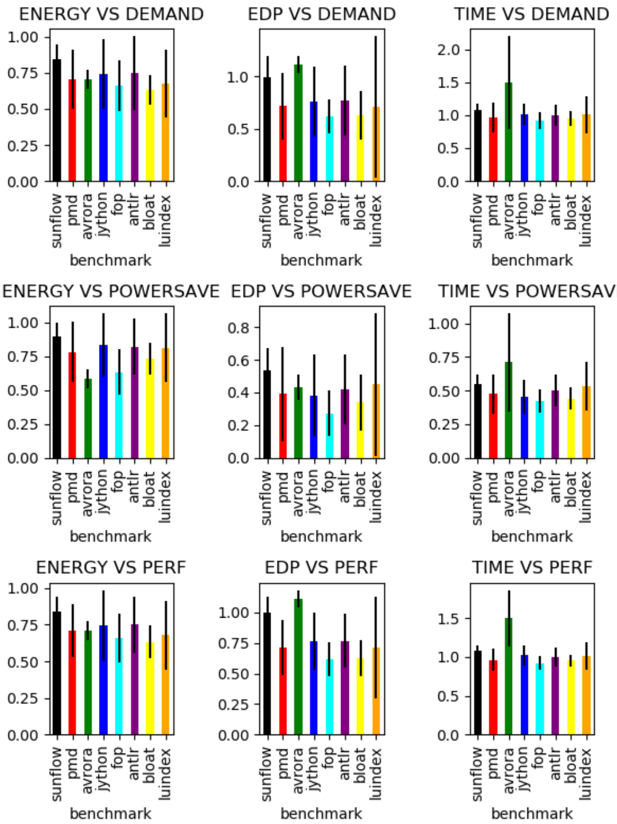


Figure 9 VINCENT Best Results against Different Governor Baselines for the First 5-Runs (All legends are otherwise identical to Fig. 8).

P-States are available, the highest power state is used. Note however the highest power state is not tantamount to the highest CPU frequency [31, 1]. Recall that (§ 2) P-States are managed at the level of the CPU package, not at the level of individual cores. How the supply voltage and the CPU frequencies of individual cores are assigned given a power state subjects to a variety of design constraints, such as area power and thermal considerations. The DVFS of VINCENT however is targeted at the core level: when a method is determined to run with the highest CPU frequency, the CPU core hosting the thread in which the method runs is set at the highest CPU frequency. This interesting phenomenon may indicate a potential for performance optimization, but there are caveats. First, the average performance improvement is small: only a subset of benchmarks can benefit, while there is degradation in others (Fig. 8). Second, as P-State maintenance is a platform-dependent black-box hardware feature, the phenomenon may be restricted to specific architectures (Broadwell in our case), and may no longer presents itself in other architectures.

5.3 The Impact during the Warm-Up Phase

The data we have shown so far result from the last 15 runs in a 20-run execution for each benchmark (see § 4), i.e., the post-warmup runs. This evaluation choice is in sync with the general focus of energy optimization on long-running applications, where energy consumption matters the most. In those server-class settings, a `sunflow` application will

continuously process images (instead of a fixed number necessitated by the benchmark), and an xalan application will continuously process XML documents (instead of a fixed number of documents).

For completeness, we now describe the result of the first 5 runs in a 20-run execution, with the per-benchmark results shown in Fig. 9. Overall, VINCENT remains an effective optimizer relative to the 3 baselines. Nearly all benchmarks retain the similar trend as post-warmup runs in Fig. 8. Relative to the latter however, the results exhibit a larger deviation. As the majority of hot methods are identified in the earlier runs, the combined 5-run results shown here demonstrate that VINCENT has already started to play an effective role in the optimization. Note however, the hot method selection process in JVMs is incremental: some hot methods may be identified during the first run, whereas others may be deferred to the later runs. As a result, the effectiveness of VINCENT relative to the 3 baselines is only incrementally more pronounced, leading to larger deviation across the 5 runs.

5.4 Multi-Method Optimization

As a part of the design space optimization, we further constructed experiments where multiple methods are subject to DVFS at the same time. Concretely, for benchmarks that have at least two methods that show favorable EDP configurations (normalized EDP < 1), we pick two methods whose least EDPs among all configurations are the smallest. We perform DVFS of both methods at the same time, adjusting the frequencies according to their respective "least EDP" configurations.

Unfortunately, the results do not show improvement. In fact, the 3 most promising benchmarks (i.e., with multiple EDP<1 configurations spanning different methods as shown in Fig. 6), pmd, avrora, and fop produced normalized EDP as 2.01, 1.77, and 1.60, respectively. The root cause is that when multiple methods are subjected to DVFS at the same time, the chance of concurrent DVFS requests increases significantly. As CPU hardware must serialize DVFS requests – DVFS is implemented as blocking I/O writes – an extensive increase in execution time ensues, bad news for energy efficiency. The multi-method result is a reminder that an overdesign may hamper effectiveness. VINCENT, as it turns out, is most effective when we keep it simple: method-grained energy optimization with a focus on the most impactful method in an application.

5.5 An Experimental Summary

Fig. 10 summarizes the average of VINCENT normalized energy/EDP/time against different baselines, across all benchmarks. On average, VINCENT can reduce energy consumption by 14.9%, EDP by 21.1%, and execution time by 12.5% against the ONDEMAND baseline. Its relative effectiveness against the POWERSAVE baseline is even more dramatic, with an EDP reduction of 63.0%. The drastic frequency downscaling in POWERSAVE may save *power*, but it is ineffective in energy optimization. On average, VINCENT's performance is on par with the PERFORMANCE baseline, with a negligible execution time reduction of 2.5%. Its effectiveness in energy and EDP reduction is similar to the result against the ONDEMAND baseline.

5.6 The Technical Report

As we described earlier, all experimental results are based on the setting where each optimization sampling interval is set at 8ms, and within each interval, 16 samples are taken. In the technical report [37], we present results with alternative sampling settings.

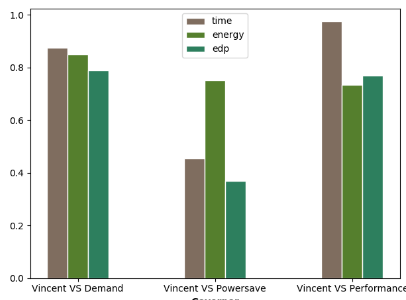


Figure 10 A Summary of Results with Different Governor Baselines (In each group, the energy/EDP/time data are normalized with their corresponding data under a built-in governor based on dynamic monitoring. For all bars, being shorter means VINCENT is more effective than the built-in governor).

The results are generally stable when the same benchmark is optimized under different sampling settings. The report also contains a discussion on the lessons we learned through the development process.

6 Related Work

Compiler-Directed or Runtime-Directed DVFS

The underlying philosophy of our work – *programs* matter for DVFS-based energy optimization – is shared among a number of compiler-directed energy optimization approaches. Saputra et al. [50] describes a DVFS-based approach at the level of compiler optimization. Their algorithm first observes the potential speed-up of loop transformation (e.g., tiling and loop fusion) over the unoptimized program, and then scales the CPU voltage and frequency down over the optimized program to a desirable level that matches the original execution time of the unoptimized program, through integer linear programming. Hsu and Kremer [28] defines a compiler-directed DVFS algorithm where a desirable CPU frequency is selected for running a code region; the selection is based on solving a minimization problem where the need for limited performance loss is encoded as constraints. Xie et al. [59] defines an analytical model – built in the compilation process – where energy minimization is reduced to a mixed-integer linear programming problem. Overall, the previous work focused on building *analytical models* in the presence of DVFS. This general direction, building analytical models to identify slacks in programs, can be traced back to a classic analysis for energy-efficient OS scheduling [57].

A small body of work further extends analytical models to virtual machines and dynamic compilation. In Haldar et al. [24], methods are instrumented with DVFS calls, and the frequency of choice when a method executes is based on the comparison among the projected energy consumption of the method at different frequencies. To make this decision, it was necessary for their analytical algorithm to introduce heuristics (that may no longer hold for state-of-the-art application workloads), such as the projected future execution time is the same as the execution time so far, and the execution time increases linearly with the CPU frequency slowdown. Wu et al. [58] proposed a dynamic compilation framework for C programs, where important code regions such as loops are manually identified and instrumented, and the CPU frequency for DVFS is selected based on an analytical model. Relative to Haldar et al., their model addressed the non-linear effect of DVFS on execution

time: through analyzing the memory-related instructions in the code region, their algorithm projects smaller performance loss for memory-intensive code regions when the CPU frequency is scaled down.

As both Haldar et al. and Wu et al. are runtime-level efforts, a more in-depth comparison is warranted. First, VINCENT does not rely on an analytical model to estimate or extrapolate the execution time or energy effect of DVFS, and does not need to instantiate the often unknown parameters in the analytical model through heuristics. Second, VINCENT identifies the most energy-consuming methods in an automated process. In contrast, the code region for DVFS in Wu et al. is manually identified, Third, both existing efforts centrally relied on instrumenting method boundaries for DVFS calls. Acceptable performance may be achievable at the era of these developments – e.g., Haldar et al. was evaluated against the Java Grande benchmark suite [54] and Wu et al. against SPEC 95 and SPEC2K – but modern Java applications are significantly more complex than e.g., `heapsort` in Java Grande. In § 3.2.2, we described the high overhead of that approach for Dacapo benchmarks.

In the context of related work, VINCENT can be understood as a revisit to a historically significant research direction – compiler/runtime-based DVFS – which has unfortunately been overtaken by black-box approaches e.g., DVFS based on dynamic performance counters. VINCENT defines an end-to-end approach that is *simple* (no analytical model), *automated* (no manual efforts in code region identification), and *scalable* in overhead (no instrumentation for DVFS). It is our hope that VINCENT is a new beginning to re-study this largely overlooked direction in the presence of modern applications in managed runtimes.

Energy-Aware Languages

Another direction of energy optimization at the boundary of programming abstractions is energy-aware programming languages [55, 10, 49, 26, 40, 19, 11, 34, 25, 41, 61, 15]. For example, Energy Types [19] introduces DVFS at the boundary of methods based on phase information declared by programmers or inferred by the compiler. Green [10] and LAB [34] select alternative algorithm-specific parameters based on energy and QoS need. Ent [14] relies on hybrid type checking to select alternative programming abstractions (methods and objects) for message dispatch. VINCENT works with the existing programming model of Java; it is an effort on runtime design instead of programming model design.

Runtime-Level Energy Efficiency

Chen et al. [18] relies on garbage collection tuning to save memory system energy consumption in JVMs. Cao et al. [16] improves the energy efficiency of JVM by assigning JVM services to small cores on asymmetric hardware. `DEP+BURST` [2] is a performance predictor and energy management system where JVM features such as synchronization, inter-thread dependencies, and store bursts, are taken into account for performance/energy prediction. Hussein et al. [29] investigates the energy impact of garbage collector design in the Android runtime. They proposed some extensions to improve the energy efficiency of asynchronous GC in Android. Overall, a common theme in existing work is to focus on JVM services (such as GC and thread management), but none considers energy optimization at the granularity of programming abstractions. Our work complements existing work with a fine-grained method-based approach for energy optimization. For unmanaged language runtimes, Hermes [47, 39] and Aequitus [48] are energy-efficient solutions built on top of Cilk. They perform DVFS based on the dependencies between thief threads and victim threads in work stealing runtimes.

Empirical studies often illuminate the energy consumption (and performance) of managed language runtimes. An early study by Vijaykrishnan et al. [56] focuses on the energy consumption impact on the memory hierarchy (cache and main memory) by JIT-enabled Java applications. Esmaeilzadeh et al. [21] studies energy efficiency with a focus on diverse configurations of workload and hardware. Sartor and Eeckhout [51] illuminates the performance of Java applications, with a focus on mapping Java application threads and JVM threads to multi-core hardware. Despite that their focus is on performance, DVFS is extensively used in their design space exploration, such as running GC threads at different CPU frequencies. Pinto et al. [45] studies the impact of energy consumption when alternative thread management designs in Java are used, such as different settings of the thread pool. Specific to ForkJoin [35], previous studies [44] also explored the impact of work stealing on the performance and energy trade-off in Java runtimes. The energy impact of different choices of Java collection classes were also a subject of studies [23, 46]. Kambadur et al. [33] takes a cross-layer approach to surveying the energy management solutions, studying the interface and interaction of different hardware/OS/compiler configurations.

Energy Profiling

Energy profiling is more commonly conducted at the system level (e.g., [43, 22]), rather than at the boundary of programming abstractions such as methods. Chappie [9] supports method-grained energy profiling. It adopts an approach with fixed time intervals, a necessary design choice when there is no JVM modification. Vincent is fundamentally a JVM-centric approach. It takes advantage of the JVM support such as instrumentation to enable delimited sampling. To Vincent, energy profiling is an intermediate step for energy optimization, which Chappie does not support.

7 Threats to Validity

While we believe leveraging hot methods in the JVM for DVFS-guided energy optimization is a generalizable idea, Vincent as an experimental system is implemented and evaluated within specific software/hardware environments. The validity of our experimental data is restricted to these environments.

First, Vincent is an extension to the JikesRVM, so the validity of our results can only be safely confirmed in that JVM. We are hopeful that the ideas behind Vincent can translate to alternative JVMs, for several reasons. (1) Vincent does not rely on unique JikesRVM features; hot method selection, dynamic instrumentation and compilation, and counter-based sampling are available in many JVMs; (2) To the best of our knowledge, alternative JVMs widely in use today do not perform DVFS-specific optimizations, so the likelihood of feature intervention is small if the idea behind Vincent is adopted on them. (3) JikesRVM has incubated other influential JVM ideas (e.g., JIT, garbage collection), whose effectiveness has been confirmed in alternative JVMs.

Second, Vincent relies on CPU architectures where DVFS is enabled. Fortunately, DVFS is a standard feature whose support is the rule not the exception in commodity CPUs, including the vast majority of chips from Intel, AMD, ARM, and others. RAPL is used for Vincent energy measurement, a hardware feature also widely available in Intel after 2011, and more recently, AMD CPUs.

Third, the experimental results are limited to the benchmark suite we used, Dacapo. Dacapo is commonly used for Java evaluating the performance of JVMs and Java applications. The benchmarks we used are multi-threaded, and they have diverse workload characteristics (CPU-bound vs. I/O-bound) that matter to energy optimization.

As for the OS governor support, note that the ONDEMAND, PERFORMANCE and POWERSAVE governors are used for the purpose of evaluation. The only OS requirement for Vincent is that the OS can expose the capability of DVFS regulation to the application. This is the USERSPACE governor in Linux. Such support is also available in other OS such as Windows [36].

8 Conclusion

Vincent is a method-grained energy optimizer residing inside the JVM. It identifies the top energy-consuming methods in the Java runtime, and performs profile-directed optimization guided by DVFS. Our experiments show Vincent can reduce the energy consumption and improve the energy efficiency of Java applications. Vincent is a novel instance among a small number of energy optimization approaches that take advantage of the information available to the managed runtime. It requires no modification to the underlying OS/hardware, and requires no programmer effort.

References

1 Kristen Accardi. Balancing power and performance in the linux kernel, https://events.static.linuxfound.org/sites/events/files/slides/LinuxConEurope_2015.pdf. In *The 2015 Linux Conference*, 2015.

2 S. Akram, J. B. Sartor, and L. Eeckhout. Dep+burst: Online dvfs performance prediction for energy-efficient managed language execution. *IEEE Transactions on Computers*, 66(4):601–615, 2017. doi:10.1109/TC.2016.2609903.

3 Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in java. *SIGPLAN Not.*, 34(10):314–324, October 1999. doi:10.1145/320385.320418.

4 Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, 2000. doi:10.1147/sj.391.0211.

5 The Linux Kernel Archives. Intel p-state driver, https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt.

6 The Linux Kernel Archives. Linux cpufreq governors, https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

7 M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization*, pages 51–62, 2005.

8 Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeño jvm. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/353171.353175.

9 Timur Babakol, Anthony Canino, Khaled Mahmoud, Rachit Saxena, and Yu David Liu. Calm energy accounting for multithreaded java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 976–988, 2020.

10 Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10*, pages 198–209, 2010.

11 Thomas Bartenstein and Yu David Liu. Green streams for data-intensive software. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, May 2013.

12 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1167473.1167488.

13 T.D. Burd and R.W. Brodersen. Energy efficient cmos microprocessor design. In *HICSS'95*, pages 288–297 vol.1, 1995.

14 Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 217–232, 2017.

15 Anthony Canino, Yu David Liu, and Hidehiko Masuhara. Stochastic energy optimization for mobile GPS applications. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 703–713, 2018.

16 Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 225–236, USA, 2012. IEEE Computer Society.

17 Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low power cmos digital design. *IEEE JOURNAL OF SOLID STATE CIRCUITS*, 27:473–484, 1995.

18 G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM Trans. Embed. Comput. Syst.*, pages 27–55, November 2002.

19 Michael Cohen, Haitao Steve Zhu, Senem Ezgi Emgin, and Yu David Liu. Energy types. In *OOPSLA '12*, 2012.

20 Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM. doi:10.1145/1840845.1840883.

21 Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 319–332, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950365.1950402.

22 X. Gao, D. Liu, D. Liu, H. Wang, and A. Stavrou. E-android: A new energy profiling tool for smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 492–502, June 2017.

23 Irene Lizeth Manotas Gutiérrez, Lori L. Pollock, and James Clause. SEEDS: a software engineer's energy-optimization decision support framework. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 503–514, 2014.

24 Vivek Haldar, Christian W. Probst, Vasanth Venkatachalam, and Michael Franz. Virtual-machine driven dynamic voltage scaling. Technical report, In Technical Report No.03-21, SICS, 2003.

25 Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214, 2015.

26 Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS '11*, 2011.

27 M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, 1994.

28 Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI'03*, pages 38–48, 2003.

29 Ahmed Hussein, Mathias Payer, Antony L. Hosking, and Christopher A. Vick. Impact of GC design on power and performance for android. In Dalit Naor, Gernot Heiser, and Idit Keidar, editors, *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, pages 13:1–13:12. ACM, 2015.

30 Intel. Energy analysis user guide, available at `https://www.intel.com/content/www/us/en/develop/documentation/energy-analysis-user-guide/`.

31 Intel. Intel 64 and ia-32 architectures software developer's manual: Volume 3, available at `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html`.

32 Canturk Isci and Margaret Martonosi. Identifying program power phase behavior using power vectors. In *In Workshop on Workload Characterization*, 2003.

33 Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 329–344, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2660193.2660196`.

34 Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA '13*, pages 661–676, 2013.

35 Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. Association for Computing Machinery. `doi:10.1145/337449.337465`.

36 Bin Lin, Arindam Mallik, Peter Dinda, Gokhan Memik, and Robert Dick. User- and process-driven dynamic voltage and frequency scaling. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 11–22, 2009. `doi:10.1109/ISPASS.2009.4919634`.

37 Kenan Liu, Khaled Mahmoud, Joonhwan Yoo, and Yu David Liu. Vincent: Green hot methods in the JVM (technical report), available at `http://www.cs.binghamton.edu/~davidl/papers/ECOOP22Long.pdf`.

38 Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *FASE 2015*, April 2015.

39 Yu David Liu. Green thieves in work stealing. In *Proceedings of ASPLOS'12 (Provactive Ideas session)*, 2012.

40 Yu David Liu. Variant-frequency semantics for green futures. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'12)*, 2012.

41 Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 575–585, 2015.

42 Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020. `doi:10.1126/science.aba3758`.

43 Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, 2012.

44 Gustavo Pinto, Anthony Canino, Fernando Castor, Guoqing (Harry) Xu, and Yu David Liu. Understanding and overcoming parallelism bottlenecks in forkjoin applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 765–775, 2017.

45 Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA '14*, 2014.

46 Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. A comprehensive study on the energy efficiency of java thread-safe collections. In *International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

47 Haris Ribic and Yu David Liu. Energy-efficient work-stealing language runtimes. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 513–528, 2014.

48 Haris Ribic and Yu David Liu. AEQUITAS: coordinated energy management across parallel applications. In *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*, pages 4:1–4:12, 2016.

49 A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI'11*, 2011.

50 H. Saputra, M. Kandemir, N. vijaykrishan, M Irwin, J. Hu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *In Proc. ACM SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems*, pages 2–11. ACM Press, 2002.

51 Jennfer B. Sartor and Lieven Eeckhout. Exploring multi-threaded java application performance on multicore hardware. In *OOPSLA'12*, OOPSLA '12, pages 281–296, 2012.

52 Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

53 Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

54 L.A. Smith, J.M. Bull, and J. Obdrizalek. A parallel java grande benchmark suite. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 6–6, 2001. doi:10.1145/582034.582042.

55 Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys '07*, pages 161–174, 2007.

56 N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, JVM 2001*, Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, JVM 2001. USENIX Association, 2001. Funding Information: This research is supported in part by grants from NSF CCR-0073419, Pittsburgh Digital Greenhouse and Sun Microsystems.; 1st Java Virtual Machine Research and Technology Symposium, JVM 2001 ; Conference date: 23-04-2001 Through 24-04-2001.

57 Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

58 Qiang Wu, V.J. Reddi, Youfeng Wu, Jin Lee, D. Connors, D. Brooks, M. Martonosi, and D.W. Clark. A dynamic compilation framework for controlling microprocessor energy and performance. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12 pp.–282, 2005. `doi:10.1109/MICRO.2005.7`.

59 Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *PLDI'03*, pages 49–62, 2003.

60 Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *In Proceedings of the USENIX Annual Technical Conference*, pages 43–56, 2003.

61 Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *ICSE'15*, pages 767–777, 2015.

Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

Amir Shaikhha
University of Edinburgh, UK

Mahdi Ghorbani
University of Edinburgh, UK

Hesam Shahrokhi
University of Edinburgh, UK

Abstract

Sets and maps are two essential collection types for programming used widely in data analytics [4]. The underlying implementation for both are normally based on 1) hash tables or 2) ordered data structures. The former provides (average-case) constant-time lookup, insertion, and deletion operations, while the latter performs these operations in a logarithmic time. The trade-off between these two approaches has been heavily investigated in systems communities [3].

An important class of operations are those dealing with two collection types, such as set-set-union or the merge of two maps. One of the main advantages of hash-based implementations is a straightforward implementation for such operations with a linear computational complexity. However, naïvely using ordered dictionaries results in an implementation with a computational complexity of $O(n \log(n))$.

Motivating Example. The following C++ code computes the intersection of two sets, implemented by `std::unordered_set`, a hash-table-based set:

```
std::unordered_set<K> result;
for(auto& e : set1) {
  if(set2.count(e))
       result.emplace(e);
}
```

However, the same fact is not true for ordered data structures; changing the dictionary type to `std::set`, an ordered implementation, results in a program with $O(n \log(n))$ computational complexity. This is because both the `count` (lookup) and `emplace` (insertion) methods have logarithmic computational complexity. As a partial remedy, the standard library of C++ provides an alternative insertion method that can take linear time, if used appropriately. The `emplace_hint` method takes a hint for the position that the element will be inserted. If the hint correctly specifies the insertion point, the computational complexity will be amortized to constant time.[1]

```
std::set<K> result;
auto hint = result.begin();
for(auto& e : set1) {
  if(set2.count(e))
       hint = result.emplace_hint(hint, e);
}
```

However, the above implementation still suffers from an $O(n \log(n))$ computational complexity, due to the logarithmic computational complexity of the lookup operation (`count`) of the second set. Thanks to the orderedness of the second set, one can observe that once an element is looked up, there is no longer any need to search its preceding elements at the next iterations. By leveraging this feature, we can provide a *hinted* lookup method with an amortized constant run-time.

[1] https://www.cplusplus.com/reference/set/set/emplace_hint/

© Amir Shaikhha, Mahdi Ghorbani, and Hesam Shahrokhi;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 33; pp. 33:1–33:3
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Hinted Data Structures. The following code, shows an alternative implementation for set intersection that uses such hinted lookup operations:

```
hinted_set<K> result;
hinted_set<K>::hint_t hint = result.begin();
for(auto& e : set1) {
  hinted_set<K>::hint_t hint2 = set2.seek(e);
  if(hint2.found)
    hint = result.insert_hint(hint, e);
  set2.after(hint2);
}
```

The above *hinted set* data-structure enables faster insertion and lookup by providing a cursor through a *hint object* (of type `hint_t`). The `seek` method returns the hint object `hint2` pointing to element `e`. Thanks to the invocation of `set2.after(hint2)`, the irrelevant elements of `set2` (which are smaller than `e`) are no longer considered in the next iterations. The expression `hint2.found` specifies if the element exists in `set2` or not. Finally, if an element exists in the second set (specified by `hint2.found`), it is inserted into its correct position using `insert_hint`.

The existing work on efficient ordered dictionaries can be divided into two categories. First, in the imperative world, there are C++ ordered dictionaries (e.g., `std::map`) with *limited* hinting capabilities only for insertion through `emplace_hint`, but not for deletion and lookup, as observed previously. Second, from the functional world, Adams' sets [1] provide efficient implementations for set-set operators. Functional languages such as Haskell have implemented ordered sets and maps based on them for more than twenty years [5]. Furthermore, it has been shown [2] that Adams' maps can be used to provide a parallel implementation for balanced trees such as AVL, Red-Black, Weight-Balanced, and Treaps. However, Adams' maps do not expose any hint-based operations to the programmer. At first glance, these two approaches seem completely irrelevant to each other.

The key contribution of this paper is hinted dictionaries, an ordered data structure that unifies the techniques from both imperative and functional worlds. The essential building block of hinted dictionaries are *hint objects*, that enable faster operations (than the traditional $O(\log n)$ complexity) by maintaining a pointer into the data structure. The underlying representation for hinted dictionaries can be sorted arrays, unbalanced trees, and balanced trees by sharing the same interface. In our running example, alternative data structures can be provided by simply changing the type signature of the hinted set from `hinted_set` to another implementation, without modifying anything else.

2012 ACM Subject Classification Software and its engineering → Functional languages; Theory of computation → Data structures design and analysis

Keywords and phrases Functional Collections, Ordered Dictionaries, Sparse Linear Algebra

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.33

Category Extended Abstract

Related Version *Full Version*: https://arxiv.org/abs/2206.04380

Acknowledgements The authors would like to thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh.

References

1 Stephen Adams. Efficient sets – a balancing act. *JFP*, 3(4):553–561, 1993.

2 Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *SPAA'16*, pages 253–264, 2016.

3 Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

4 Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *PACMPL*, 6(OOPSLA1):1–33, 2022.

5 Milan Straka. The performance of the haskell containers package. *ACM Sigplan Notices*, 45(11):13–24, 2010.

Slicing of Probabilistic Programs Based on Specifications

Marcelo Navarro
Computer Science Department (DCC), University of Chile, Santiago, Chile

Federico Olmedo
Computer Science Department (DCC), University of Chile, Santiago, Chile

Abstract

We present the first slicing approach for probabilistic programs *based on specifications*. Concretely, we show that when probabilistic programs are accompanied by their functional specifications in the form of pre- and post-condition, one can exploit this semantic information to produce specification-preserving slices *strictly more precise* than slices yielded by conventional techniques based on data/control dependency.

To illustrate this, assume that Alice and Bob repeatedly flip a fair coin until observing a matching outcome, either both heads or both tails. However, Alice decides to "trick" Bob and switches the outcome of her coin, before comparing it to Bob's. The game can be encoded by the program below, which is instrumented with a variable n that tracks the required number of rounds until observing the first match. The program terminates after K loop iterations with probability $1/2^K$ provided $K > 0$, and with probability 0 otherwise, satisfying the annotated specification.

$\backslash\backslash$ *pre:* $\frac{1}{2^K}[K > 0]$
$n \coloneqq 0;$
$a, b \coloneqq 0, 1;$
while $(a \neq b)$ do
  $n \coloneqq n + 1;$
  $\{a \coloneqq 0\}\ [1/2]\ \{a \coloneqq 1\};$
  $a \coloneqq 1 - a;$
  $\{b \coloneqq 0\}\ [1/2]\ \{b \coloneqq 1\}$
$\backslash\backslash$ *post:* $[n = K]$

Traditional slicing techniques based on data/control dependencies conclude that the only valid slice of the program (w.r.t. output variable n) is the very same program. However, our slicing approach allows removing the assignment $a \coloneqq 1 - a$ from the loop body, while preserving the program specification.

At the technical level, our slicing technique works by propagating post-conditions backward using the greatest pre-expectation transformer – the probabilistic counterpart of Dijkstra's weakest pre-condition transformer. This endows programs with an axiomatic semantics, expressed in terms of a verification condition generator (VCGen) that yields quantitative proof obligations.

In particular, we design (and prove sound) VCGens for both the partial (allowing divergence) and the total (requiring termination) correctness of probabilistic programs, making our slicing technique *termination-sensitive*. To handle iteration, we assume that program loops are annotated with invariants. To reason about (probabilistic) termination, we assume that loop annotations also include (probabilistic) variants.

Another appealing property of our slicing technique is its *modularity*: It yields valid slices of a program from valid slices of its subprograms. Most importantly, this involves only *local reasoning*.

Besides developing the theoretical foundations of our slicing approach, we also exhibit an algorithm for computing program slices. Interestingly, the algorithm computes the *least* slice that can be derived from the slicing approach, according to a proper notion of slice size, using, as main ingredient, a shortest-path algorithm.

Finally, we demonstrate the applicability of our approach by means of a few illustrative examples and a case study from the probabilistic modeling field.

© Marcelo Navarro and Federico Olmedo;
licensed under Creative Commons License CC-BY 4.0
36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 34; pp. 34:1–34:2
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2012 ACM Subject Classification Theory of computation → Probabilistic computation; Theory of computation → Program specifications; Software and its engineering → Designing software

Keywords and phrases probabilistic programming, program slicing, expectation transformer semantics, verification condition generator

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.34

Category Extended Abstract

Related Version *Full Version*: https://doi.org/10.1016/j.scico.2022.102822

Funding This research has been supported by the FONDECYT Grant No. 11181208.

Prisma: A Tierless Language for Enforcing Contract-Client Protocols in Decentralized Applications

David Richter
Technische Universität Darmstadt, Germany

David Kretzler
Technische Universität Darmstadt, Germany

Pascal Weisenburger
Universität St. Gallen, Switzerland

Guido Salvaneschi
Universität St. Gallen, Switzerland

Sebastian Faust
Technische Universität Darmstadt, Germany

Mira Mezini
Technische Universität Darmstadt, Germany

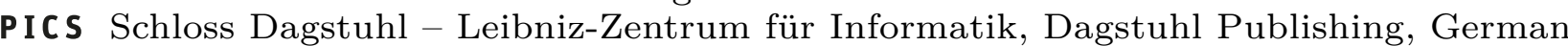
Abstract

Decentralized applications (dApps) consist of smart contracts that run on blockchains and clients that model collaborating parties. dApps are used to model financial and legal business functionality. Today, contracts and clients are written as separate programs – in different programming languages – communicating via send and receive operations. This makes distributed program flow awkward to express and reason about, increasing the potential for mismatches in the client-contract interface, which can be exploited by malicious clients, potentially leading to huge financial losses. In this paper, we present Prisma, a language for tierless decentralized applications, where the contract and its clients are defined in one unit. Pairs of send and receive actions that "belong together" are encapsulated into a single direct-style operation, which is executed differently by sending and receiving parties. This enables expressing distributed program flow via standard control flow and renders mismatching communication impossible. We prove formally that our compiler preserves program behavior in presence of an attacker controlling the client code. We systematically compare Prisma with mainstream and advanced programming models for dApps and provide empirical evidence for its expressiveness and performance.

The design space of dApp programming and other multi-party languages depends on one major choice: a *local* model versus a *global* model. In a *local* model, parties are defined in separate programs and their interactions are encoded via send and receive effects. In a *global* language, parties are defined within one shared program and interactions are encoded via combined send-and-receive operations with no effects visible to the outside world. The global model is followed by tierless [18, 8, 4, 10, 24, 25, 19, 26] and choreographic [12, 15, 11] languages. However, known approaches to dApp programming follow the local model, thus rely on explicitly specifying the client–contract interaction protocol. Moreover, the contract and clients are implemented in different languages, hence, developers have to master two technology stacks. The dominating approach in industry is Solidity [14] for the contract and JavaScript for clients. Solidity relies on expressing the protocol using assertions in the contract code, which are checked at run time [1]. Failing to insert the correct assertions may give parties illegal access to monetary values to the detriment of others [16, 13]. In research, contract languages [9, 5, 22, 23, 7, 6, 17, 3] have been proposed that rely on advanced type systems such as session types, type states, and linear types. The global model has not been explored for dApp programming. This is unfortunate given the potential to get by with a standard typing discipline and to avoid intricacies and potential mismatches of a two-language stack. Our work fills this gap by proposing Prisma – the first language that features a *global programming model* for Ethereum dApps. While we focus on the Ethereum blockchain, we believe our techniques

© David Richter, David Kretzler, Pascal Weisenburger, Guido Salvaneschi, Sebastian Faust, and Mira Mezini; licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 35; pp. 35:1–35:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



to be applicable to other smart contract platforms. Prisma enables interleaving contract and client logic within the same program and adopts a *direct style (DS)* notation for encoding send-and-receive operations (with our `awaitCl` language construct) akin to languages with async/await [2, 21]. DS addresses shortcomings with the currently dominant encoding of the protocol's *finite state machines (FSM)* [14, 5, 22, 23, 7, 6]. We argue writing FSM style corresponds to a control-flow graph of basic blocks, which is low-level and more suited to be written by a compiler than by a human. With FSM style, the contract is a passive entity whose execution is driven by clients. whereas the DS encoding allows the contract to actively ask clients for input, fitting dApp execution where a dominant contract controls execution and diverts control to other parties when their input is needed.

In the following Prisma snippet, the `payout` function is a function invoked by the contract when it is time to pay money to a client. In Prisma, variables, methods and classes are separated into two namespaces, one for the contract and one for the clients. The `payout` method is located on the contract via the annotation `@co`. The body of the method diverts the control to the client using `awaitCl(...) { ... }`, hence the contained `readLine` call is executed on the client. Note that no explicit send/receive operations are needed but the communication protocol is expressed through the program control flow. Only after the check `client == toBePayed` that the correct client replied, the current contact balance `balance()` is transferred to the client via `transfer`.

```
1 @co def payout(toBePayed: Arr[Address]): Unit = {
2   awaitCl(client => client == toBePayed) {
3     readLine("Press enter for payout") }
4   toBePayed.transfer(balance())
5 }
```

Overall, Prisma relieves the developer from the responsibility of correctly managing distributed, asynchronous program flows and the heterogeneous technology stack. Instead, the burden is put on the compiler, which distributes the program flow by means of selective continuation-passing-style (CPS) translation and defunctionalisation and inserts guards against malicious client interactions.

We needed to develop a CPS translation for the code that runs on the Ethereum Virtual Machine (EVM) since the EVM has no built-in support for concurrency primitives which could be used for asynchronous communication. While CPS translations are well-known, we cannot use them out-of-the-box because the control flow is interwoven with distribution in our case. A CPS translation that does not take distribution into account would allow malicious clients to force the contract to deviate from the intended control flow by sending a spoofed continuation. Thus, it was imperative to prove correctness of our *distributed CPS translation* to ensure control-flow integrity of the contract.

2012 ACM Subject Classification Software and its engineering → Distributed programming languages; Software and its engineering → Domain specific languages; Software and its engineering → Compilers

Keywords and phrases Domain Specific Languages, Smart Contracts, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.35

Category Extended Abstract

Related Version *Technical Report*: https://arxiv.org/abs/2205.07780 [20]

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.8.2.16
Software (Source Code): https://github.com/stg-tud/prisma

Funding This work has been funded by the German Federal Ministry of Education and Research *iBlockchain project* (BMBF No. 16KIS0902), by the German Research Foundation (DFG, SFB 1119 *CROSSING Project*), by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity *ATHENE*, by the Hessian LOEWE initiative (*emergenCITY*), by the Swiss National Science Foundation (SNSF, No. 200429), and by the University of St. Gallen (IPF, No. 1031569).

References

1. Solidity documentation - common patterns. `https://docs.soliditylang.org/en/v0.7.4/common-patterns.html`, 2020. Accessed 14-11-2020.
2. Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause 'n' play: Formalizing asynchronous c#. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 233–257. Springer, 2012. `doi:10.1007/978-3-642-31057-7_12`.
3. Sam Blackshear, Evan Cheng, D. Dill, Victor Gao, B. Maurer, T. Nowacki, Alistair Pott, S. Qadeer, Dario Russi, Stephane Sezer, Tim Zakian, and Run tian Zhou. Move: A language with programmable resources, 2019.
4. Kwanghoon Choi and Byeong-Mo Chang. A theory of RPC calculi for client–server model. *Journal of Functional Programming*, 29, 2019.
5. Michael J. Coblenz. Obsidian: a safer blockchain programming language. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 97–99. IEEE Computer Society, 2017. `doi:10.1109/ICSE-C.2017.150`.
6. Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Jonathan Aldrich, Joshua Sunshine, and Brad A. Myers. User-centered programming language design in the obsidian smart contract language. *CoRR*, abs/1912.04719, 2019. `arXiv:1912.04719`.
7. Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *CoRR*, abs/1909.03523, 2019. `arXiv:1909.03523`.
8. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06, pages 266–296, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1777707.1777724`.
9. Ankush Das, S. Balzer, J. Hoffmann, and F. Pfenning. Resource-aware session types for digital contracts. *ArXiv*, abs/1902.06056, 2019.
10. Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *Proceedings of the ACM on Programming Languages*, 3(POPL):28:1–28:29, January 2019. `doi:10.1145/3290341`.
11. Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects, 2020. `arXiv:2005.09520`.
12. Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega K. Ojo, editors, *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, volume 6536 of *Lecture Notes in Computer Science*, pages 55–75. Springer, 2011. `doi:10.1007/978-3-642-19056-8_4`.
13. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2976749.2978309`.
14. Mix. These are the top 10 programming languages in blockchain. `https://thenextweb.com/hardfork/2019/05/24/javascript-programming-java-cryptocurrency/`, 2019. Accessed 14-11-2020.
15. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014. `doi:10.1007/978-1-4614-7518-7_4`.

16 Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 653–663, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3274694.3274743.

17 Reed Oei, Michael J. Coblenz, and Jonathan Aldrich. Psamathe: A DSL with flows for safe blockchain assets. *CoRR*, abs/2010.04800, 2020. arXiv:2010.04800.

18 Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 23–33. ACM, 2000. doi:10.1145/351240.351243.

19 Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In Atsushi Igarashi, editor, *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS '16, pages 377–397, Berlin, Heidelberg, November 2016. Springer-Verlag. doi:10.1007/978-3-319-47958-3_20.

20 David Richter, David Kretzler, Pascal Weisenburger, Guido Salvaneschi, Sebastian Faust, and Mira Mezini. Prisma: A tierless language for enforcing contract-client protocols in decentralized applications (extended version), 2022. doi:10.48550/ARXIV.2205.07780.

21 Scala. Scala async rfc. http://docs.scala-lang.org/sips/pending/async.html.

22 Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in flint. In Stefan Marr and Jennifer B. Sartor, editors, *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, pages 218–219. ACM, 2018. doi:10.1145/3191697.3213790.

23 Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. Flint for safer smart contracts. *CoRR*, abs/1904.06534, 2019. arXiv:1904.06534.

24 Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA Companion '06, New York, NY, USA, 2006. ACM.

25 Manuel Serrano and Vincent Prunet. A glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, pages 180–192, New York, NY, USA, 2016. ACM. doi:10.1145/2951913.2951916.

26 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):129:1–129:30, October 2018. doi:10.1145/3276499.